UNIVERSITÉ DE MONTRÉAL

ANALYSE DÉTAILLÉE DE TRACE EN DÉPIT D'ÉVÉNEMENTS MANQUANTS

MARIE MARTIN
DÉPARTEMENT DE GÉNIE INFORMATIQUE ET GÉNIE LOGICIEL
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

MÉMOIRE PRÉSENTÉ EN VUE DE L'OBTENTION
DU DIPLÔME DE MAÎTRISE ÈS SCIENCES APPLIQUÉES
(GÉNIE INFORMATIQUE)
AOÛT 2018

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Ce mémoire intitulé :

ANALYSE DÉTAILLÉE DE TRACE EN DÉPIT D'ÉVÉNEMENTS MANQUANTS

présenté par : MARTIN Marie
en vue de l'obtention du diplôme de : Maîtrise ès sciences appliquées
a été dûment accepté par le jury d'examen constitué de :

Mme BELLAÏCHE Martine, Ph. D., présidente
M. DAGENAIS Michel, Ph. D., membre et directeur de recherche
M. BILODEAU Guillaume-Alexandre, Ph. D., membre

DÉDICACE

À ma famille.
Corinne, Fabien, Manon et Maxence,
rien n'aurait été possible sans vous....

REMERCIEMENTS

J'aimerais avant tout remercier mon directeur de recherche, Michel Dagenais, sans lequel je n'aurais jamais pu découvrir le monde de la recherche. Ses précieux conseils ont su répondre à mes doutes et mes hésitations tout au long de ce projet, et son expérience et son impressionnante culture scientifique ont, sans aucun doute, été des sources d'inspiration essentielles. Merci de m'avoir accueillie au sein du DORSAL.

Un grand merci également à Geneviève Bastien, pour son aide inestimable durant ce projet et pour avoir toujours été disponible en cas de besoin. Sa maîtrise de Trace Compass et ses excellentes idées ont été d'un grand secours. Merci à Naser Ezzati-Jivan pour ses conseils et retours lors de la rédaction de l'article.

Je voudrais aussi remercier tous mes collègues du DORSAL. La passion et la bonne humeur de chacun d'entre eux créent une atmosphère propice à la recherche. Ils ont tous su rendre cette maîtrise plus agréable. Une mention spéciale à Robin pour les excellentes pauses-café que j'ai passées avec lui. Nos discussions ont permis de me faire progresser dans les moments difficiles.

Je remercie le CRSNG (Conseil de Recherches en Sciences Naturelles et en Génie du Canada), le Conseil Régional de Nord Pas de Calais-Picardie, le Ministère de l'Enseignement Supérieur et de la Recherche, et tous les partenaires du laboratoire, Ericsson, Ciena, EfficiOS, Google et Prompt, pour leur soutien financier.

RÉSUMÉ

Le traçage offre une compréhension détaillée du fonctionnement d'un système ou d'une application, mais la simple étude d'une trace ne permet pas d'exploiter tout le potentiel des informations contenues dans les événements qui la constituent. C'est pour cela que les spécialistes des systèmes distribués et de l'analyse de performance développent des analyses complexes, qui sont ensuite intégrées aux outils de visualisation de traces.

Ces outils supposent que le processus de traçage s'est déroulé sans erreurs. Néanmoins, il arrive que des événements soient perdus. Ceci survient lorsque les structures de données (souvent des tampons circulaires) devant stocker les événements avant leur écriture dans la trace sur disque sont pleins, alors que d'autres événements sont générés. Pour éviter de bloquer le système, le traceur doit jeter les événements les plus récents, ou écraser les plus anciens. Il n'existe pas de mécanisme pour gérer ce cas lors de l'analyse de trace, ce qui crée des incohérences dans les résultats sans que l'utilisateur en soit notifié.

L'objectif de cette recherche est donc d'étudier le problème des événements perdus dans une trace lors de son analyse, et d'apporter des solutions permettant d'indiquer les incohérences et d'y remédier. Notre hypothèse était que les événements présents dans la trace contiennent suffisament d'information pour pouvoir détecter un changement d'état qui n'aurait pas dû avoir lieu, et pour compléter la chronologie manquante. Pour cela, nous avons utilisé des machines à états finis, puisque ce formalisme de représentation permet de modéliser le système étudié, en définissant la façon dont les événements génèrent des changements d'états. Ces machines à états possèdent des propriétés qui peuvent être exploitées pour retrouver les informations perdues.

L'analyse de traces se fait séquentiellement, en lisant la trace événement par événement. D'une part, la machine à états est mise à jour après chaque événement traité, si celui-ci provoque une transition. À ce moment-là, nous pouvons vérifier si le nouvel état atteint est un état certain, c'est-à-dire que l'on sait avec certitude que l'état généré est effectivement l'état réel dans lequel était le système. C'est une propriété utile pour savoir si l'on peut se fier aux résultats de l'analyse. Lorsqu'on atteint un intervalle pendant lequel des événements perdus sont rapportés, tous les états deviennent incertains, car il est possible que les informations manquantes aient empêché des changements d'états de se produire. D'autre part, si aucune transition n'est possible de l'état actuel, nous vérifions si l'événement n'aurait pas dû en provoquer une depuis un autre état. Si c'est le cas, nous détectons une incohérence, car cela signifie que nous avons manqué des transitions vers cet état.

Après la collecte des incohérences, une phase de correction peut avoir lieu. Pour chaque incohérence, nous calculons la séquence de transitions qui aurait pu être exécutée entre le dernier état cohérent connu et l'état incohérent détecté. Nous utilisons pour cela l'algorithme de Dijkstra pour calculer le chemin le plus court entre ces deux états (la machine à états pouvant être assimilée à un graphe). Les transitions (arcs du graphe) sont pondérées par des statistiques sur la fréquence des transitions prises. De la séquence de transitions déduites découle l'inférence de la séquence d'événements correspondante. Puis, pour chaque événement déduit, nous essayons de compléter son contenu en calculant des valeurs possibles pour les champs supposés, à partir des conditions qui doivent être validées pour l'exécution de la transition associée.

Nous avons implémenté cette solution dans l'outil de visualisation de traces Trace Compass. Les informations concernant les incertitudes et les incohérences sont visuellement ajoutées à la vue présentant les résultats de l'analyse. Puis, si l'utilisateur le souhaite, il peut visualiser une autre vue qui prend en compte les événements déduits.

Notre solution a été appliquée à plusieurs cas d'utilisation, permettant ainsi d'apporter des précisions à l'analyse et de récupérer des événements perdus. La précision de la correction est de 53%. Nous l'avons testée avec plusieurs traces de tailles différentes. Le surcoût apporté est raisonnable pour des traces de petites tailles, mais il peut rapidement augmenter si les traces sont très grandes, avec beaucoup d'incohérences. Dans tous les cas, l'inférence d'événements n'est pas très coûteuse (moins de 8% dans le pire des cas).

Nous avons donc montré qu'il était possible de récupérer des événements perdus à partir d'une trace incomplète et d'une machine à états modélisant le système tracé. Nous sommes capable d'indiquer si un état est certain ou incertain, et notre solution peut détecter les incohérences par rapport à la définition d'une machine à états. Ces résultats ouvrent la voie à des développements dans le domaine de l'analyse de traces en parallèle, en réduisant les dépendances entre morceaux de traces, et en permettant la reconstruction d'un état initial sans avoir à analyser séquentiellement toute la trace.

ABSTRACT

With tracing, a lot of information can be gathered from a system or an application. This information needs to be exploited in order to provide a useful insight into the operation of our system. Experts develop complex trace analyses, but it does not take into account the fact that some events may have been lost during the tracing process, due to a much bigger flow of generated events than the capacity of writing into the trace. Therefore, the results may contain some inconsistencies.

Our objective is to deal with these lost events at the trace analysis level. Using finite state machines to model the system, we are able to check the certainty of each new state. If no transition can be triggered from the current state, we check if one could be triggered from another state. By doing so, we can find inconsistencies. Then, we can correct these by looking for the sequence of transitions between the last coherent state and the incoherent one. Dijkstra's shortest-path algorithm is applied, with frequencies used as weights for the transitions. Once the transitions have been inferred, we can deduce the related lost events and their content.

The proposed solution has been implemented in Trace Compass, a trace visualisation tool. The new information is displayed on the view representing the results of the analysis. It allowed us to apply our method to a few usecases, thus demonstrating its usefulness. The accuracy of the recovery phase is 53%. The overhead of the detection phase is reasonable for small traces, but can increase rapidly for bigger ones. However, the inference phase does not cost too much in any case.

With this work, we successfully demonstrated that lost events can be recovered in many cases, using an incomplete trace and a state machine modelling the system, that inconsistencies can be detected and that we can evaluate the certainty of a state. These results show the way forward for the parallel analysis of traces, because we can reconstruct an initial state without sequentially analysing a whole trace.

TABLE DES MATIÈRES

LISTE DES TABLEAUX

LISTE DES FIGURES

LISTE DES SIGLES ET ABRÉVIATIONS

DES	Discrete Event System
FSM	Finite State Machine
LTTng	Linux Trace Toolkit : next generation
OTF	Open Trace Format
CPU	Centralized Processing Unit
API	Application Programming Interface
CTF	Common Trace Format
XML	Extensible Markup Language
DORSAL	Laboratoire de recherche sur les systèmes répartis ouverts et très disponibles
MSC	Message Sequence Chart
TCP	Transmission Control Protocol

LISTE DES ANNEXES

CHAPITRE 1 INTRODUCTION

Les systèmes répartis sont aujourd'hui largement répandus, que ce soit dans le domaine des télécommunications, de l'infonuagique, ou encore du calcul à haute performance. L'étude de tels systèmes nécessite une compréhension détaillée de leur fonctionnement et des mécanismes internes qui les régissent. En enregistrant les événements bas-niveau qui ont lieu dans le système à chaque instant, le traçage permet de répondre à ce défi.

L'analyse de trace permet ensuite de mettre à profit les informations brutes contenues dans la trace. Celle-ci est lue séquentiellement, et chaque événement est traité pour mettre à jour le système d'états géré par le module d'analyse. Mais que se passe-t-il lorsque des événements sont manquants dans la trace ? On comprend facilement que des incohérences vont apparaître dans les résultats de l'analyse.

D'autre part, les machines à états sont un formalisme de représentation que l'on utilise pour modéliser un système. Puisqu'il décrit le fonctionnement du système considéré, on peut avoir l'intuition qu'il est possible de retrouver les événements perdus en utilisant un modèle global du comportement attendu et les événements disponibles concernant l'exécution du système. C'est pourquoi nous nous y intéresserons particulièrement dans ce travail.

1.1 Définitions et concepts de base

1.1.1 Événement

Un événement est un élément de base d'une trace. D'un point de vue général, il est constitué d'un type qui l'identifie, d'une estampille de temps, et d'un ensemble d'informations qui forme son contenu. Ces informations sont représentées sous forme de champs, avec un nom et une valeur (par exemple, `(cpu, 1)` pour indiquer que l'événement a eu lieu sur le CPU 1).

Il est le moyen de représenter une action concrète qui a eu lieu sur le système à un instant t, et qui va donc lui être associée. Il est généré lorsqu'un point de trace est atteint. Un événement peut aussi être synthétique, c'est-à-dire être construit à partir d'événements bruts contenus dans la trace (Ezzati-Jivan and Dagenais, 2012), afin d'obtenir une représentation plus haut-niveau d'une action sur le système.

1.1.2 Traçage

Le traçage est une technique consistant à enregistrer tous les événements se produisant dans un système, comme la boîte noire d'un avion. Ces événements sont enregistrés dans une trace par un outil qu'on appelle un traceur. Par la suite, la trace peut être lue afin d'obtenir une séquence ordonnée des événements : ouverture d'un fichier, écriture, exécution d'un processus, fermeture d'un fichier, etc. Cette tâche de lecture est réalisée par un analyseur de traces, dont un utilisateur peut se servir pour diagnostiquer le système tracé, notamment dans le but de trouver la cause de problèmes de performance.

1.1.3 Analyse de traces

L'analyse de traces permet de générer des rapports graphiques ou textuels sur l'activité du système pendant la trace (Reumont-Locke, 2015). Cela consiste donc à extraire d'une trace les informations utiles selon le but recherché par l'utilisateur. On comprend qu'il existe un grand nombre d'analyses variées : utilisation des CPUs, de la mémoire, statistiques sur la fréquence des types d'événements, états des processus, analyse du chemin critique, etc.

1.1.4 Vue

Une vue est une représentation graphique des résultats d'une analyse particulière. En général, une vue est un graphe XY (graphique linéaire, histogramme) ou une frise chronologique (temps en abscisse). Elle offre une vue d'ensemble, et permet à l'utilisateur de saisir facilement les résultats. Il est possible de naviguer sur la vue, en zoomant, sélectionnant un intervalle de temps, filtrant des objets spécifiques...

1.2 Éléments de la problématique

Le traçage permet de trouver la cause de problèmes qui n'auraient pas pu être étudiés avec d'autres méthodes, comme le profilage ou le débogage. À ce titre, c'est une technique précieuse pour analyser des systèmes complexes. Malheureusement, il arrive que lors du traçage, certains événements soient perdus, car le flux d'événements se produisant sur le système est trop important par rapport à la vitesse maximale d'écriture dans la trace (e.g., sur disque ou à travers un lien réseau). Dans ce cas, le traceur doit jeter les `n` événements qui arrivent alors que le tampon est plein, et est seulement capable d'enregistrer dans la trace l'information « `n` événements sont perdus ». Lors de l'analyse, il va donc manquer certaines informations (qui étaient contenues dans les événements jetés) afin d'obtenir un diagnostic correct. Ac-

tuellement, l'analyseur n'est pas capable de traiter l'absence d'événements, il va simplement réaliser l'analyse comme si aucun événement n'était manquant, ce qui a pour conséquence de fausser les résultats. Il est très important de réaliser un diagnostic avec seulement des informations correctes, ou du moins, l'utilisateur doit être en mesure de juger de l'exactitude du diagnostic, en sachant qu'il est possible que certaines portions de la trace soient incomplètes.

Le problème vient du fait qu'il n'est pas possible d'empêcher la perte d'événements, du fait des choix d'architecture fixés pour le traceur. Toute la difficulté vient du fait qu'il faut être capable de récupérer des informations suffisamment précises pour être exploitables et ne pas biaiser l'analyse. Il n'existe actuellement aucune prise en compte des événements manquants pendant l'analyse des traces. Par conséquent, le problème considéré explore un nouveau champ qui n'avait pas encore été considéré. Comme nous allons le montrer à la section 2, il n'existe pas de solution dans la littérature concernant l'analyse de traces. Les solutions proposées dans d'autres domaines sont difficilement applicables au cas considéré, et elles n'exploitent pas les informations supplémentaires dont on dispose dans une trace.

De manière générale, toute application gérant des flux de données est succeptible de perdre une partie des informations transmises. C'est par exemple le cas des paquets perdus sur le réseau, si l'on considère un protocole de communication. L'application de mesures visant à juger de la pertinence de l'analyse va donc au-delà du cas d'utilisation originel, le traçage.

Outre la problématique des événements perdus, un important cas d'étude est celui de l'analyse de traces en parallèle. Cette méthode consiste à diviser une trace en plusieurs sections, afin d'analyser parallèlement chacune des sections. Cela permet d'améliorer la performance de l'analyse. Toutefois, le principal obstacle réside dans le fait que l'analyse se base sur une lecture séquentielle des événements, et le résultat au temps t dépend du résultat au temps $t-1$. Si un morceau donné de trace commence au temps t, on peut considérer que les données au temps $t-1$ sont des données perdus. Ainsi, la solution proposée pour répondre à la problématique des événements perdus pourrait être appliquée pour supporter l'analyse de traces lorsqu'on ne connaît pas l'état initial.

Comment des données peuvent-elles être manquantes ? Dans LTTng (et plusieurs autres traceurs), les événements sont enregistrés dans un tampon circulaire, constitué de plusieurs sous-tampons. Lorsqu'un sous-tampon est plein, un démon consomme les événements de ce sous-tampon et les écrit dans la trace (voir 2.1.2 pour plus de détails).

Concernant les données manquantes, il y a plusieurs cas de figure à considérer :

— le tampon circulaire qui collecte les événement est plein, et les nouveaux événements sont jetés ;

— le tampon circulaire qui collecte les événement est plein, et les événements du tampon sont ré-écrits par les nouveaux événements ;

— une panne provoque la perte d'un événement (non-enregistrement) ;

— les données proviennent d'un cliché (*snapshot*) du contenu des tampons de traçage, donc les événements précédents ne sont pas connus ;

— les données sont traitées en parallèle, on n'a pas accès aux événements précédents et suivants l'intervalle de données considéré.

Les deux premiers cas correspondent aux deux modes de fonctionnement du canal (*event loss mode*) dans LTTng [1] :

— *discard mode* : lorsque le tampon est plein, les nouveaux événements qui arrivent sont jetés et un compteur est incrémenté pour chaque événement jeté (celui-ci sera ensuite écrit dans la trace) ;

— *overwrite mode* : lorsque le tampon est plein et que de nouveaux événements arrivent, le sous-tampon contenant les événements les moins récents est ré-écrit. Contrairement au *discard mode*, c'est donc un ensemble d'événements qui est jeté (l'entièreté du contenu du sous-tampon). Un compteur est incrémenté du nombre d'événements que contenaient ce sous-tampon.

Grâce au compteur mentionné ci-dessus, il est possible de connaître le nombre d'événements perdus (qu'ils aient été ré-écrits ou jetés).

La disparition de certains événements implique que l'on manque des transitions dans la FSM, puisque l'on ne rencontre pas les événements qui auraient dû les provoquer. On se retrouve alors dans un nouvel état qui ne devrait normalement pas être accessible depuis l'état courant. L'état courant devient incohérent.

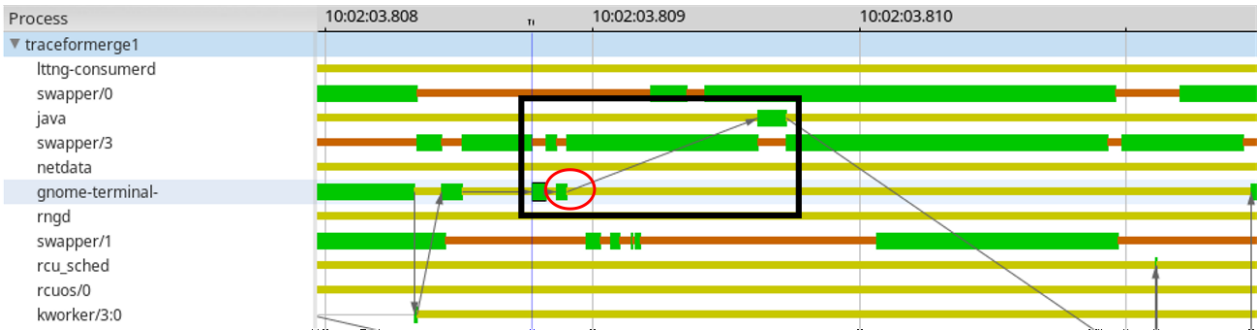
Voici un exemple : un thread p_1 est en cours d'exécution (état *running*). Un événement e_1 *sched_switch* est reçu (c'est-à-dire, lu dans la trace), avec *next_tid* = *swapper* et *prev_tid* = p_1. Cela a pour effet de modifier l'état de p_1 à attente (*waiting*). L'événement e_2 suivant est également un *sched_switch* avec *next_tid* = p_2 et *prev_tid* = *swapper*, ce qui provoque la transition de p_2 à l'état exécution. p_2 reste dans cet état jusqu'à la réception d'un événement *sched_switch* où *prev_tid* est p_2. Plus tard, un événement (que l'on appellera e_3) *sched_switch* avec *next_tid* égal à p_1 sera reçu et changera à nouveau l'état de p_1 en exécution.

À présent, l'événement e_1 est manuellement supprimé de la trace. L'état de p_1 ne sera donc pas modifié de exécution à attente, puisque c'est la réception et le traitement de l'événement qui

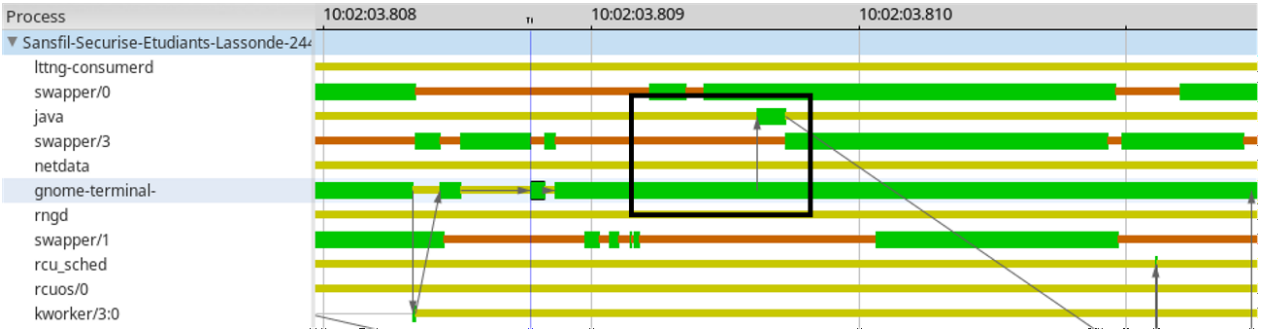
1. https ://www.lttng.org/docs/v2.8/#doc-channel-overwrite-mode-vs-discard-mode

provoque normalement cette transition. p_1 reste dans l'état exécution. Toutefois, l'événement e_2 se produit, ce qui modifie l'état de p_2 à exécution. On a ici un premier état incohérent, puisque 2 threads ne peuvent pas être actifs en même temps, sur le même cpu. Plus tard, e_3 provoque un autre état incohérent : en effet, cet événement devrait provoquer le passage à l'état exécution pour p_1, or ce dernier est déjà dans cet état. Cela prouve à nouveau que p_1 est dans un état incohérent.

Ces observations sont visibles sur la figure 1.1, avec p_1 le thread *gnome_terminal* et p_2 le thread *java*, tels que représentés dans Trace Compass. L'événement qui a été supprimé est entouré en rouge sur la figure 1.1a.



(a) Trace sans événement manquant.



(b) Trace avec un événement manquant.

Figure 1.1 Exemple de l'analyse d'une trace normale et de la même trace avec un événement manquant

1.3 Objectifs de recherche

La problématique générale de cette recherche peut être formulée comme suit :

> Comment peut-on supporter des événements manquants lors de l'analyse d'une trace ?

L'objectif final du projet est de pallier le problème des événements manquants du mieux possible, c'est-à-dire de donner le plus d'outils possible à l'utilisateur afin de juger de la pertinence de l'analyse produite. Idéalement, la solution proposée doit permettre de retrouver les événements perdus.

Afin d'atteindre l'objectif général, plusieurs objectifs spécifiques peuvent être formulés :

— Avoir une définition rigoureuse d'un événement et des informations nécessaires que celui-ci doit contenir, ainsi que de la machine à états qui utilise ces événements, afin de correctement analyser les traces ;

— Être capable de reconnaître et d'indiquer des événements incohérents, par rapport à la machine à états qui correspond au système tracé, et dont la présence témoigne de la perte d'événements antérieurs ;

— Pouvoir retrouver des éléments d'information sur les événements perdus à partir d'autres événements de la trace et des spécifications de la machine à états ;

— Proposer un format de représentation des événements perdus ou incohérents, ainsi que du degré d'incertitude (quant à la cohérence de l'état global ou à la fiabilité des informations déduites), et utiliser ce format pour présenter visuellement l'information pertinente à l'utilisateur (comme par exemple la présence d'un état probable, ou la présence d'un état global incohérent pour une certaine période) ;

— Intégrer la solution à l'analyseur de traces existant.

1.4 Plan du mémoire

Dans le chapitre 2, nous présenterons la revue de littérature traitant des travaux importants pour le sujet étudié. En association avec le chapitre 3, cela servira à introduire des pistes de solution pour notre problématique. Le cœur de ce mémoire se trouve dans l'article présenté au chapitre 4. La solution proposée y est décrite, ainsi que des cas d'utilisation et l'évaluation. Le chapitre 5 permettra de discuter de l'article présenté au chapitre précédent et d'approfondir certains concepts. Nous conclurons dans le chapitre 6.

CHAPITRE 2 REVUE DE LITTÉRATURE

2.1 Traçage

Nous allons commencer par introduire la technique fondamentale sur laquelle repose ce travail, à savoir le traçage. Bien que nous nous situions à l'étape suivante (*i.e.*, l'analyse de traces), il est important de saisir les principes du traçage, puisqu'ils sont à la base de notre problématique. Nous allons donc présenter les concepts essentiels, puis quelques-uns des outils existants et leurs particularités.

2.1.1 Généralités

Le traçage est une technique de collecte d'informations sur un système, utilisée pour diagnostiquer des cas complexes, comme des problèmes de performance (par exemple, voir (Nemati et al.)). Les informations collectées sont l'ensemble des événements bas-niveau ayant lieu dans le système (il est à noter qu'il est possible de filtrer certains événements, dans le cas où ils ne sont d'aucun intérêt pour l'utilisateur). On peut citer, entre autres, les événements d'ordonnancement, les appels système, la manipulation de fichiers, etc.

Il existe d'autres techniques de diagnostic, les plus courantes étant le profilage et le débogage. Le profilage consiste à instrumenter le code source ou l'exécutable d'un programme puis à l'exécuter afin de récolter des mesures sur son fonctionnement, tel que le nombre d'appels pour chaque fonction, la quantité de mémoire utilisée pour chaque type d'objets, etc. L'idée derrière cela est d'identifer des mesures aberrantes qui vont diriger vers les endroits dans le code source à améliorer pour bénéficier d'un gain de performance pour l'application. Quant au débogage, c'est une technique permettant d'exécuter un programme instruction par instruction, ou de le stopper à des endroits particuliers du code (points d'arrêt) et d'étudier la pile mémoire, la pile d'appels, etc. L'objectif est d'identifier la cause des bogues, puis de les corriger.

Le traçage se distingue donc du profilage et du débogage par le fait qu'il répond à la fois à la problématique d'identification d'un problème (profilage, par exemple identifier qu'une certaine méthode est très lente), mais aussi à celle de l'identification de la cause (débogage, par exemple identifier une variable mal initialisée). Le traçage se base sur la collecte d'informations très précises sur le système, ce qui permet ensuite de réaliser des analyses avancées sur la trace. Notamment, il est possible de trouver la cause des problèmes étudiés, et non pas seulement leur emplacement, grâce à l'historique fourni par la trace (Fournier and Dagenais). Cette technique est donc très utile pour comprendre le fonctionnement des systèmes distri-

bués, dont l'analyse est rendue difficile du fait des caractéristiques mêmes de ces systèmes. De plus, le traçage introduit un plus faible surcoût que la plupart des profileurs. Or, certains problèmes ne sont observables qu'en cas d'exécution réelle du système (par exemple, un bogue sensible à l'ordre d'exécution d'instructions parallèles). Ils ne peuvent pas être étudiés à l'aide d'un débogueur classique, car celui-ci introduit un trop fort surcoût qui va modifier le comportement du programme lors de l'exécution (Toupin, 2011).

Afin de tracer un système, il faut que des événements soient enregistrés, ce qui a lieu lorsqu'un point de trace est atteint et exécuté. Il existe deux manières distinctes d'insérer des points de trace : l'instrumentation statique, et l'instrumentation dynamique (Fahem). La première technique consiste à définir les points de trace dans le code avant la compilation. Par la suite, ils peuvent être activés (génération d'un événement quand le point de trace est atteint) ou désactivés (no-op). C'est une technique plus performante, qui offre un surcoût presque nul quand le système n'est pas tracé, c'est-à-dire que les points de trace sont désactivés (mais ce surcoût existera toujours, même si le point de trace est désactivé). Par contre, il en découle qu'il est plus difficile d'insérer de nouveaux points de trace puisque cela nécessite de recompiler le programme, et il faut donc avoir accès au code source de l'application à tracer. L'instrumentation dynamique permet de pallier ce problème en se basant sur l'insertion, à n'importe quel endroit, de points de trace durant l'exécution du programme, par modification du code binaire. Elle est plus flexible, mais un point de trace dynamique implique un surcoût par rapport à un point de trace statique activé. Seuls les points de trace nécessaires sont ajoutés au programme par l'utilisateur, donc le surcoût global par rapport à l'instrumentation statique va dépendre du nombre de points de trace dynamiques ajoutés. En outre, cette technique nécessite que l'utilisateur ait une connaissance approfondie du programme.

On peut aussi faire la distinction entre le traçage noyau et le traçage utilisateur, selon l'espace mémoire considéré. Le premier a pour objectif de récupérer des informations sur l'exécution du système d'exploitation en traçant l'espace noyau. Quant au traçage utilisateur, il permet d'obtenir des informations sur une application en particulier (en traçant l'espace utilisateur) mais nécessite donc que celle-ci ait été instrumentée préalablement. Toutefois, il est en général possible d'obtenir des données sur une application à partir du traçage noyau puisque, dans la plupart des cas, une application passe par le système d'exploitation.

2.1.2 Outils existants

ftrace *ftrace*[1] (*Function Tracer*) est le traceur interne du noyau Linux introduit avec la version 2.6.27. En plus d'être un traceur, il fournit un ensemble de fonctionnalités pour

1. https ://elinux.org/Ftrace

diagnostiquer un système (modules d'analyse). Il utilise des points de trace statiques et de l'instrumentation dynamique des fonctions du noyau (Rostedt, 2008). L'interface de ftrace est accessible dans le système de fichier *tracefs* (avant la version 4.1, dans debugfs ; par rétrocompatibilité, si debugfs est monté, tracefs l'est aussi automatiquement), mais il est aussi possible d'utiliser la commande `trace-cmd` pour une interface plus conviviale.

Lors de la configuration, il est possible de choisir parmi plusieurs traceurs afin de sélectionner le traceur actuel. On peut mentionner "*function*" pour tracer les appels de fonctions, "*hwlat*" pour tracer les causes matérielles de latence, "*wakeup*" pour tracer la latence maximale qu'une tâche prend pour être ordonnancée après son réveil, etc.

Le traceur de fonctions utilise des tampons en anneau, ce qui peut mener de nouvelles données à écraser des données contenues dans le tampon, si celui-ci est en mode réécriture. Les tampons peuvent aussi être utilisés en mode producteur/consommateur (les nouvelles données sont perdues si le tampon est plein) [2]. Leur taille peut être modifiée par l'utilisateur.

Le fait que ce traceur soit intégré au noyau Linux facilite son utilisation. Néanmoins, un important désavantage est l'impossibilité d'effectuer le traçage en espace utilisateur. De plus, certains choix architecturaux par défaut limitent ses fonctionnalités : trace gardée en mémoire, taille limitée d'un événement, etc. C'est pourquoi d'autres traceurs ont été proposés.

perf *perf*[3] est un traceur inclus dans le noyau Linux à partir de la version 2.6+. Il s'utilise en ligne de commande. Cet outil utilise notamment les compteurs de performance (De Melo, 2010), qui sont des registres matériels du CPU pouvant stocker des informations comme, par exemple, le nombre de défauts de cache ou le nombre d'instructions exécutées. Ils fonctionnent en comptant le nombre des événements matériels pertinents pour chaque compteur : *cache-misses* pour les défauts de cache, *instructions* pour les instructions exécutées, et bien d'autres (*cpu-cycles, branch-misses, ...*). perf supporte aussi d'autres types d'événements : logiciels (*page-faults, context-switches, ...*), points de trace (*syscalls :sys_enter_socket* par exemple), etc.

Il peut être configuré pour le système au complet, mais aussi par CPU, par fil d'exécution, ou par tâche, grâce à l'échantillonnage basé sur les interruptions : lorsqu'un compteur dépasse une certaine valeur, une interruption est lancée et un échantillon est enregistré.

Mais perf propose aussi d'autres fonctions utiles pour diagnostiquer un système (Gregg, 2013). Citons par exemple qu'il est possible de créer dynamiquement des points de trace avec `perf probe`, qui utilise les *kprobes* et les *uprobes*. `perf report` lit les données enregistrées

2. plus de détails sur : https ://www.kernel.org/doc/Documentation/trace/ring-buffer-design.txt
3. https ://perf.wiki.kernel.org/index.php/Main_Page

après le traçage et crée un résumé des informations obtenues ; ou encore, `perf stat` permet d'afficher des statistiques essentielles sur les compteurs de performance.

Il est donc intéressant d'avoir un outil permettant de collecter des informations à la fois logicielles et matérielles, ce qui permet d'avoir un ensemble de statistiques assez facilement. Cepedant, comme *ftrace*, il n'est pas possible de tracer en espace utilisateur. Les structures de données de ce traceur sont moins optimisées, ce qui explique pourquoi ses performances sont inférieures (Gebai and Dagenais, 2018).

LTTng *LTTng*[4] est un traceur non bloquant capable de tracer le noyau Linux et l'espace utilisateur d'une application instrumentée (Desnoyers, 2009). L'objectif de ce traceur est de fournir un grand nombre d'informations sur le système, avec le plus faible coût possible, ce qui est particulièrement important pour tracer les systèmes temps réel. C'est pourquoi il est important d'être non bloquant : si le système est bloqué dès que les tampons sont pleins, on risque de perdre d'importantes propriétés sensibles au temps et à l'ordre d'exécution (par exemple, des situations de concurrence).

On peut illustrer le caractère non-bloquant de LTTng avec la figure 2.1. On y voit une application en train d'être tracée, et pour laquelle 2 événements viennent d'être générés. Chaque événement est envoyé au tampon correspondant (par exemple, chaque événement concerne un CPU différent). Dans un cas, l'événement `i` peut correctement être enregistré dans le sous-tampon actuellement disponible. Dans l'autre cas, l'événement `j` doit être jeté car tous les sous-tampons sont pleins et n'ont pas encore été consommés par le démon en charge de vider un sous-tampon pour l'écrire dans la trace. Cela pourrait arriver en cas d'explosion soudaine de l'activité sur un CPU donné, qui entraînerait la génération d'un nombre très important d'événements.

Les traces générées par LTTng sont au format CTF [5]. Ce format est optimisé pour assurer une grande performance lors de l'écriture. Une trace est divisée en plusieurs flux d'événements binaires (en général, LTTng écrit un flux par CPU), où les événements sont classés par ordre chronologique d'estampilles de temps, ainsi qu'un flux de métadonnées (des informations sur la trace, comme un identificateur unique, l'environnement utilisé lors de sa génération, les types d'événements enregistrés...). De plus, il est possible de créer plusieurs sessions et ainsi, d'écrire plusieurs traces simultanément.

La commande `lttng` permet de contrôler le traceur : l'utilisateur peut créer des sessions, les configurer en modifiant la taille des tampons, en choisissant les événements à activer, etc.,

4. https ://lttng.org/
5. http ://diamon.org/ctf/

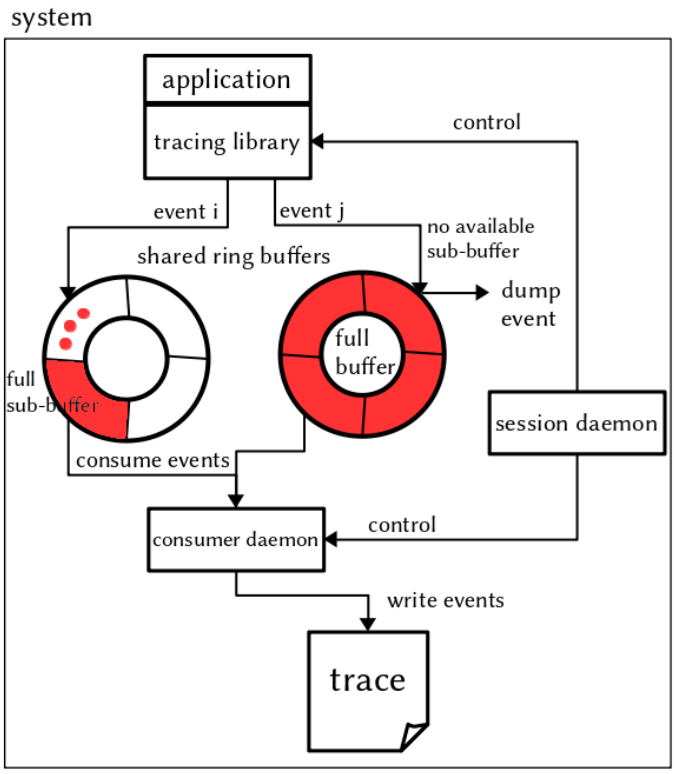


Figure 2.1 Schéma simplifié de l'architecture de LTTng, présentant le rôle des tampons circulaires et les mécanismes principaux de son fonctionnement

démarrer et stopper le traçage...

Comme mentionné plus haut, LTTng utilise des tampons pour y enregistrer les événements générés. Ces tampons sont circulaires, et chacun est constitué d'un certain nombre de sous-tampons de taille identique. Il existe deux principaux modes de fonctionnements pour les tampons :

— abandon (*discard mode*) : lorsque le tampon est plein, les nouveaux événements qui arrivent sont jetés et un compteur représentant le nombre d'événements perdus est incrémenté pour chaque événement jeté (celui-ci sera ensuite écrit dans la trace) ;

— réécriture (*overwrite mode*) : le tampon est ré-écrit lorsqu'il est plein et que de nouveaux événements arrivent ;

Depuis la version 2.10, le traceur en espace utilisateur supporte aussi un mode bloquant, mais ce n'est pas un cas d'utilisation qui nous intéresse.

LTTng présente des performances meilleures que celles de perf et ftrace (Gebai and Dagenais,

2018), avec une faible latence pour un point de trace en espace noyau. De plus, un des avantages importants est qu'il est possible de tracer à la fois l'espace noyau et l'espace utilisateur d'une application (Goulet, 2012).

Avec les travaux présentés dans Reumont-Locke (2015), on comprend qu'il y a actuellement de plus en plus de données à traiter dans une trace, notamment à cause de l'augmentation du nombre de cœurs dans les processeurs. Or, l'analyse de traces est actuellement séquentielle. La mise à l'échelle de l'analyse de traces apparaît comme nécessaire, d'où l'analyse de traces en parallèle. Pour cela, une trace est découpée en plusieurs morceaux, selon des techniques de partitionnement de données. Il y a une découpe par canal (horizontalement), et une découpe par intervalle de temps (verticalement), réalisée de telle sorte que la charge de travail est équilibrée sur chaque morceau.

Néanmoins, l'analyse de trace en parallèle pose de nombreux défis. Un de ces défis est la dépendance des données entre elles. Citons, par exemple, le manque d'accès à des informations nécessaires pour la réalisation du traitement sur une unité, dû au fait que ces informations sont contenues sur une autre unité. De plus, «l'analyse de traces est une tâche se basant sur un système d'états global »(Reumont-Locke, 2015), qui est normalement construit de façon séquentielle. La trace est lue événement par événement, en suivant l'ordre chronologique, et chaque événement est traité pour mettre à jour le ou les états concernés. Donc on comprend que, si une trace est divisée en plusieurs morceaux devant chacun être traité en parallèle, chaque morceau n'a pas accès à l'état global le précédant, puisque celui-ci ne sera construit qu'après l'exécution de l'analyse parallèle sur le morceau précédent.

Nous avons donc besoin d'une solution qui permet de réaliser une analyse sans accès à un état global. Il serait également intéressant de pouvoir dépasser le problème de dépendance des données en étant capable de déduire la valeur prise par les événements inaccessibles.

Les analyses de traces hors-ligne peuvent être représentées par des machines à états (voir Matni and Dagenais, 2009), où les transitions sont des événements. Chaque machine distincte peut être exécutée en parallèle, mais le gain est minime, car la plus grande partie de l'analyse consiste à décoder les événements.

Un système peut être modélisé par un système d'états, avec un état global qui peut être récupéré pour chaque instant dans le temps. Cet état peut être construit à partir d'une trace grâce à des attributs dont la valeur est représentée sous forme d'intervalles qui sont stockés dans un arbre d'historique d'états.

On peut donc remarquer que, quel que soit le traceur considéré, on ne peut éviter la perte

d'événements si l'on veut respecter la contrainte non-bloquante. Le problème des événements manquants est alors sensiblement le même dans tous les cas. Dans le cadre de notre travail, nous nous baserons sur LTTng pour générer les traces au format CTF.

Une fois la trace obtenue, il faut la traiter. Nous traçons un système pour obtenir des informations sur son fonctionnement, pour le diagnostiquer. On veut donc analyser la trace et observer les résultats. C'est ce que nous allons présenter dans la section suivante.

2.2 Visualisation

Nous allons présenter ici les principaux outils permettant de visualiser une trace et les résultats des analyses qui peuvent lui être appliquées. Nous soulignerons leurs caractéristiques distinctives, ainsi que leurs limitations.

2.2.1 Analyse de traces

Après avoir tracé un système, nous obtenons une trace contenant l'ensemble des événements observés durant le traçage. Les informations contenues dans la trace peuvent être exploitées afin d'obtenir diverses métriques et autres renseignements utiles sur le système (Giraldeau et al., 2011). Ce traitement correspond à la phase dite de l'analyse de traces, et a pour objectif de fournir plusieurs outils à l'utilisateur afin que celui-ci puisse diagnostiquer le système qu'il a tracé. Par exemple, un utilisateur faisant face à un programme dont le temps d'exécution est inhabituellement long, pourrait tracer ce programme et exécuter l'analyse du chemin critique dans le but d'identifier la cause du problème ; ce pourrait être une instruction causant une attente dans le programme.

2.2.2 Outils existants

L'objectif des outils de visualisation de traces est de fournir à l'utilisateur des informations avancées sur le système tracé (c'est-à-dire celles que l'on a obtenues lors de l'analyse de traces) sous une forme graphique. En effet, il est très peu probable qu'un utilisateur puisse tirer un quelconque diagnostic simplement à partir de l'étude d'une trace brute, contenant des millions d'événements. Il faut donc présenter les résultats des analyses dans un format compréhensible et simple d'utilisation, comme des graphes, des statistiques, etc.

Nous allons donc passer en revue certains des outils de visualisation de traces les plus communs.

Chromium trace-viewer *Chromium*[6] est un navigateur web à code source ouvert, basé sur le navigateur *Google Chrome*. Il propose un outil de traçage permettant d'étudier ses performances[7]. Pour chaque fil d'exécution de chaque processus de Chrome, l'outil enregistre les signatures de méthodes C++ ou Javascript. Les traces sont écrites selon le format *trace_event*. Il peut y avoir perte d'événements, puisque les tampons ont une taille limitée.

L'outil *Trace Viewer*[8] est capable de visualiser les traces Chrome au format trace_event, ainsi que des traces à d'autres formats (comme les traces noyau Linux, générées avec *ftrace*). Un des composants de l'outil s'appelle *Model* et a pour rôle de stocker les données obtenues à partir des traces. L'unité logique de base est une tranche (*slice*), constituée d'un intervalle de temps et de métadonnées sur cet intervalle. Les tranches sont construites à partir des événements lus dans la trace. Par exemple, un événement signalant le début d'une action (type B dans le format trace_event[9]) et un événement signalant la fin de celle-ci (type E) vont permettre la construction d'une tranche.

Un exemple de trace visualisée avec cet outil peut être trouvé à la figure 2.2.

Il n'existe pas de mécanisme particulier pour gérer les événements perdus, puisque ce traceur a été initialement conçu pour ne tracer que le navigateur Chrome, pour lequel les tampons en mémoire ont une taille fixe. Ce cas d'utilisation très spécifique ne nécessitait donc pas de gestion des événements perdus, c'est pourquoi le problème n'est pas adressé.

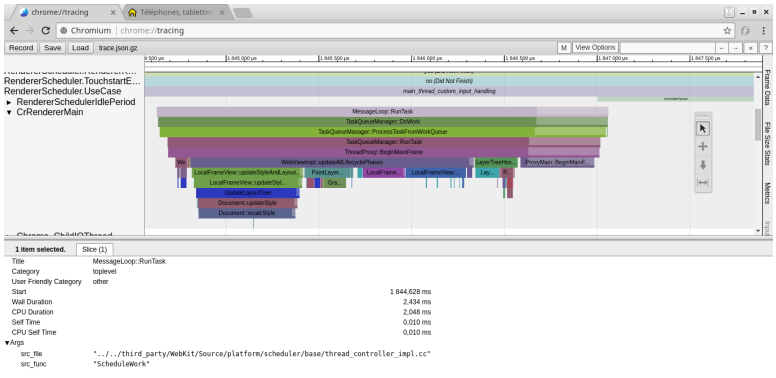


Figure 2.2 Visualisation d'une trace avec Chromium, après avoir tracé le chargement de la page web fido.ca

6. https ://www.chromium.org
7. https ://www.chromium.org/developers/how-tos/trace-event-profiling-tool
8. https ://github.com/catapult-project/catapult/tree/master/tracing
9. voir https ://docs.google.com/document/d/1CvAClvFfyA5R-PhYUmn5OOQtYMH4h6I0nSsKchNAySU/edit#heading=h.uxpopqvbjezh

Traceshark [10] est un autre outil de visualisation de traces, développé en C++. Il supporte les traces noyau Linux contenant des événements de *ftrace* et de *perf* et permet d'afficher quelques analyses de base, comme l'état des CPUs, leur fréquence, les tâches d'ordonnancement, etc., ou encore un *Flame Graph*. La figure 2.3 illustre un exemple de vue avec cet outil.

Cet outil ne supporte pas les événements perdus, comme indiqué dans la documentation :

> For both Ftrace and perf it is very desirable to avoid lost events because traceshark cannot visualize correctly with lost events, nor can it find a wakeup event that has been lost.

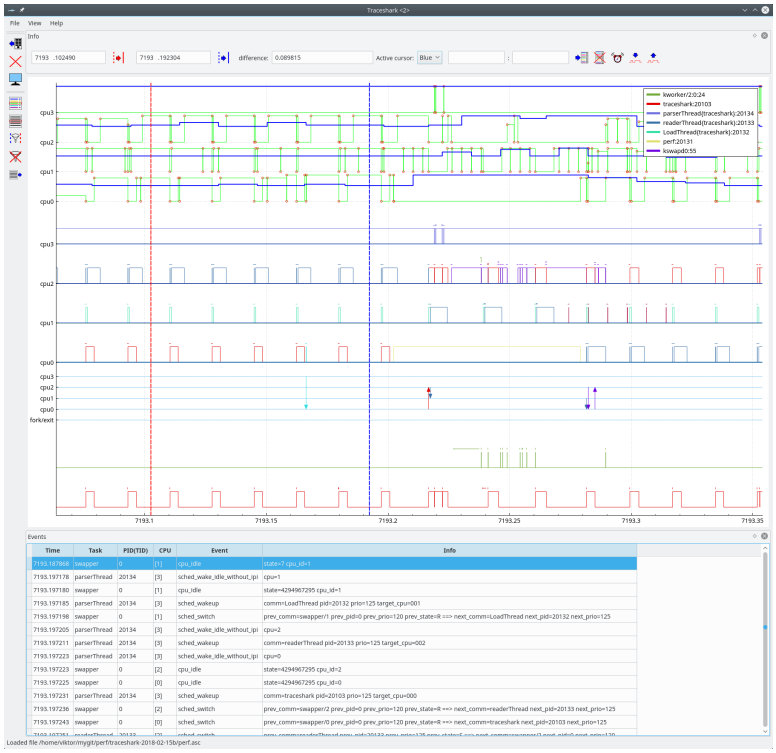


Figure 2.3 Visualisation d'une trace avec Trace Shark [11]

Trace Compass *Trace Compass* [12] est une application à code source ouvert permettant l'analyse et la visualisation de traces (notamment au format CTF, mais aussi les traces GDB, BTF, libcap pour les traces réseau, etc.). Elle est développée en Java et intégrée à l'environnement de développement Eclipse. De nombreuses analyses avancées sont mises à la disposition des utilisateurs : analyse de la latence, corrélation de traces, corrélation de

10. https ://github.com/cunctator/traceshark
12. http ://tracecompass.org/

paquets réseau, et bien d'autres. Il est également possible de déclarer ses propres analyses, dans un format basé sur le XML, que nous détaillerons plus à la partie 2.3.2. Un exemple de vue est fourni à la figure 2.4.

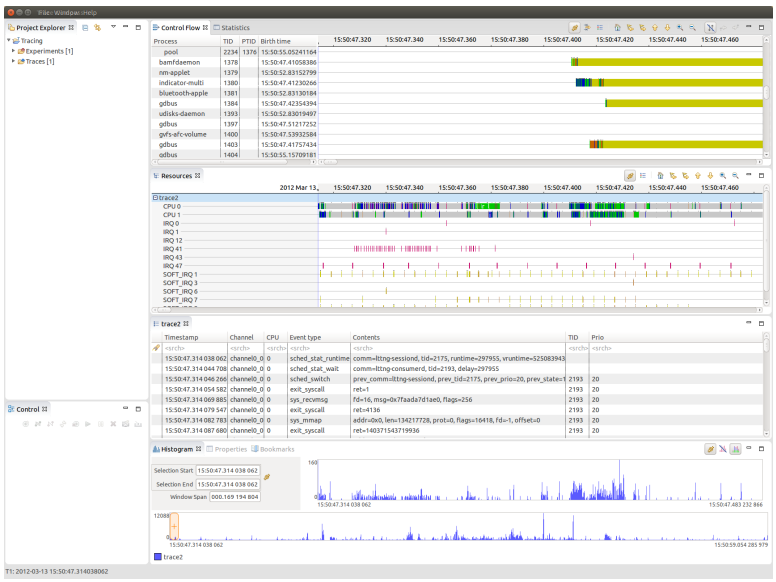


Figure 2.4 Visualisation d'une trace avec Trace Compass [13]

Trace Compass fonctionne avec un système d'états. Un état est caractérisé par un intervalle de temps (date de début et date de fin) et une valeur (valable pendant la durée de l'intervalle). Plusieurs états peuvent simultanément exister dans le système. Un attribut est caractérisé par un état unique et possède une valeur (éventuellement nulle) à n'importe quel point dans le temps. Ainsi, à un temps t déterminé, on peut connaître l'état de chaque attribut. C'est le plus petit morceau du système qui puisse avoir un état. L'état global du système est alors défini comme l'ensemble des attributs du modèle à un instant t, et on l'obtient donc en examinant l'état de chaque attribut.

Tant que l'attribut n'a pas été initialisé, sa valeur est nulle (*null*). Toutefois, il existe un événement appelé `lttng_statedump_process_state`, qui réalise une opération afin de récupérer l'état d'un processus. On utilise ce type d'événement pour construire un état initial. Ce dernier est nécessaire pour réaliser certaines analyses, car elles ont besoin de construire les états suivants, provenant des données extraites des événements, à partir d'une valeur de départ, c'est-à-dire d'un état initial (un cas trivial est celui où l'on voudrait incrémenter un compteur, ce qui nécessite de connaître la valeur initiale de ce compteur). Par exemple, on va propager `lttng_statedump_process_state` sur chaque thread, afin d'initialiser le système d'états avec la valeur de l'état de chaque thread. Cela permet ensuite d'obtenir un état global.

C'est ce qu'on appelle le cliché de l'état courant (*state dump*).

Le système d'états (*State System*) repose sur deux objets importants :

— l'arbre des attributs (*attribute tree*) : une structure de données en arbre qui permet de stocker les attributs ;

— le gestionnaire d'états (*state provider*) : il produit les changements d'état (*state changes*) à partir des événements reçus et crée les nouvelles valeurs d'états (*state values*).

Chaque analyse a une façon particulière de définir et de construire l'arbre des attributs et les états. En fait, c'est au développeur utilisant cet outil qu'incombe la tâche de définir une structure du système d'états qui correspond au modèle du système, afin d'alimenter les analyses réalisées sur les traces.

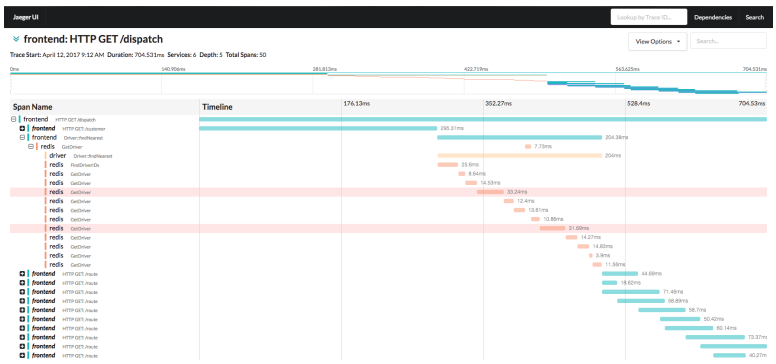


Figure 2.5 Visualisation d'une trace avec Jaeger [14]

Jaeger *Jaeger*[15] est une solution de traçage distribué à code source ouvert, développée en partenariat avec la *Cloud Native Computing Foundation*, visant les architectures de micro-services. L'application dorsale est développée en *Go*, et les clients Jaeger utilisent l'API *Open Tracing*[16]. L'interface utilisateur est développée en *React/Javascript* ; la visualisation des traces se fait donc dans un navigateur web. Un exemple de vue est présenté à la figure 2.5.

Une trace Jaeger est constituée d'intervalles (*spans*) formant un graphe acyclique orienté. Un intervalle représente une unité logique, une tâche, et est composé d'un temps de début, d'une durée, et d'un nom de tâche. Ainsi, l'analyse reçoit directement ces intervalles sans avoir à les construire à partir d'événements. En cas d'événements perdus, il n'est pas possible de travailler avec des machines à états puisque les états sont déjà construits ; ces cas ne pourraient alors pas être traités avec Jaeger.

15. https ://www.jaegertracing.io
16. http ://opentracing.io/

On constate donc que cet outil est conçu pour travailler à un niveau d'abstraction supérieur aux outils considérés précédemment, par son utilisation du concept d'intervalles. C'est un outil assez récent, qui vise des cas d'utilisation spécifiques (microservices), d'où son caractère assez spécialisé.

Autres projets Voici d'autres outils qu'il est important de mentionner, mais que nous n'étudierons pas en détail puisqu'ils reprennent des concepts déjà traités par les projets que nous avons présentés ci-dessus.

Vampir[17] est une suite d'outils permettant d'étudier la performance d'applications parallèles (MPI, OpenMP...). Le format Open Trace Format (OTF) est supporté et la version 9.4 est la plus récente à ce jour.

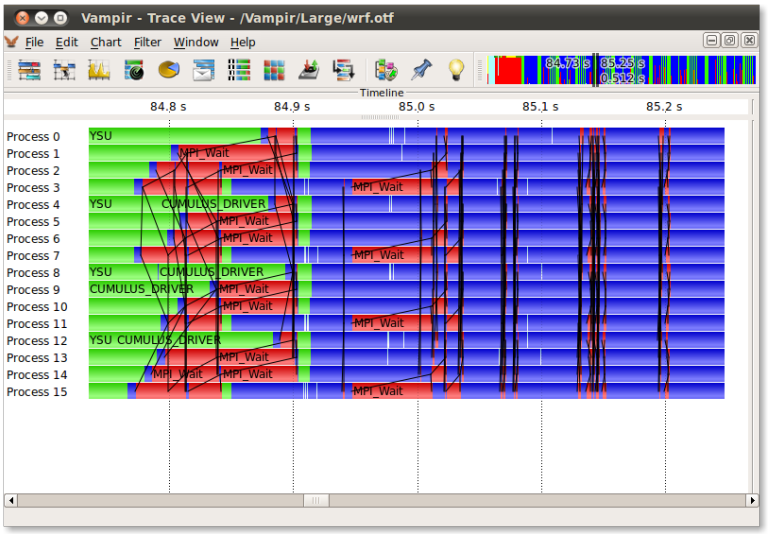


Figure 2.6 Visualisation d'une trace avec Vampir[18]

Paraver[19] est un autre outil d'analyse et de visualisation à destination des applications parallèles. Il est développé par le *Barcelona Supercomputing Center*, et se veut très flexible ; c'est pourquoi les métriques sont définies par l'utilisateur, à partir d'un ensemble de filtres, de fonctions, etc. De plus, le format de trace utilisé (format à code source ouvert propre à Paraver) n'a pas de sémantique. C'est à l'utilisateur d'utiliser les éléments à sa disposition pour transformer la trace et indiquer au module de visualisation quoi afficher. Une trace contient un en-tête et une liste d'enregistrements qui peuvent être de trois types différents : événement (un événement ponctuel à un temps donné), état (un intervalle pour un état donné d'un fil d'exécution), et lien/communication (une relation entre deux objets de la trace).

17. https ://www.vampir.eu/
19. https ://tools.bsc.es/paraver

Il n'y a donc pas de notion de machine à états avec cet outil. Ce dernier reçoit directement les états, et ne serait donc pas capable de traiter des événements perdus.

Un nombre limité de vues est disponible (voir, par exemple, la figure 2.7 pour la vue chronologique).

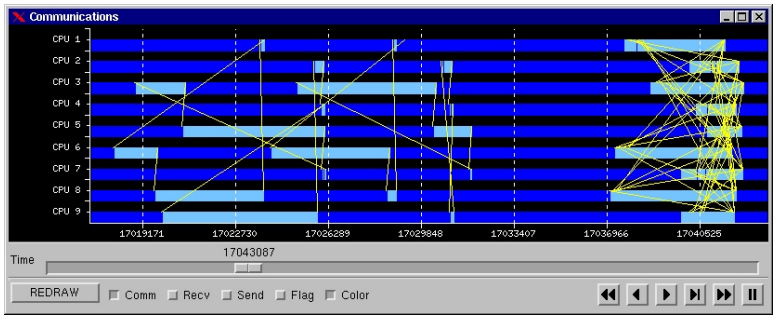


Figure 2.7 Visualisation d'une trace avec Paraver [20]

Nous pouvons classer tous ces outils en deux catégories : ceux qui construisent les états à partir des événements de la trace (Chromium, Traceshark...), et ceux qui reçoivent directement les états sous la forme d'intervalles (Jaeger, Paraver...). Selon l'objectif que l'on cherche à atteindre, seuls les outils de la première catégorie seraient capables de travailler avec des machines à états. En outre, dans le premier cas, la plupart de outils construisent un état à partir d'une paire d'événements constituant le début et la fin de l'état. Ce n'est pas le cas de Trace Compass, qui est capable de réaliser des analyses plus complexes, puisqu'il formule explicitement la machine à états utilisée pour transformer les événements en états. Nous verrons dans la section suivante comment cette machine à états est définie.

2.3 Machines à états finis

Afin de comprendre sur quoi se basent les analyses de traces, nous devons comprendre ce qu'est une machine à états finis et comment elles peuvent nous être utiles. Nous allons aborder les machines à états telles qu'elles sont définies dans Trace Compass en particulier, puisque c'est l'outil avec lequel nous avons travaillé, mais cela ne limite pas l'application des machines à états à d'autres cas d'utilisation.

2.3.1 Théorie

Les automates à états finis (*Finite State Machines*, FSM) sont un formalisme de représentation utilisé notamment pour modéliser un système. Une FSM a la particularité de pouvoir prendre ses états parmi un ensemble fini d'états possibles.

Mathématiquement, on les définit comme un quintuplé (Σ, S, s_0, δ, F) avec :

- Σ : l'alphabet d'entrée, ensemble fini non vide de symboles ;
- S : les états possibles, ensemble fini non vide ;
- s_0 : l'état initial, appartenant à S ;
- δ : la fonction de transition d'états, tel que $\delta : S \times \Sigma \to S$;
- F : les états finaux, sous-ensemble de S, qui peut être vide.

De plus, un automate est dit déterministe si (Fabre, 2013) :

$$|I| = 1$$

et

$$\forall (s, \alpha) \in S \times \Sigma, |\delta(s, \alpha)| \leq 1$$

Concrètement, cela signifique que chaque état a, au plus, une seule transition possible pour une entrée donnée parmi les entrées possibles.

Il existe un algorithme permettant de transformer toute FSM non déterministe en FSM déterministe (mais la taille de l'automate produit peut être exponentielle à la taille de l'automate de départ).

Vérifier la cohérence de l'état local d'un processus correspond à vérifier si la séquence d'événements sur le processus est reconnue par le langage généré par la FSM le modélisant. Les traces que nous avons à notre disposition sont, en général, des séquences d'événements pour le modèle global du système, et non pour un modèle local (par exemple, un processus). Ainsi, il faut être capable de distinguer les événements selon le modèle considéré, c'est-à-dire savoir à quelle machine s'applique un événement donné. En fait, dans le cas d'utilisation général, on veut une instance de FSM par ressource : si le système est constitué de 3 CPUs, on veut 3 instances de FSMs modélisant un CPU ; et si, sur les CPUs, il y a 10 threads, on veut 10 instances de FSMs modélisant un thread. Ceci est une nécessité, car il est impossible de construire un modèle global du système (cela nécessiterait une connaissance extrêmement poussée du système, un nombre gigantesque d'états et la complexité de ce modèle le rendrait inutilisable). Ceci explique la nécessité de séparer le modèle par ressource. De plus, le fait d'avoir une instance de FSM par instance de ressources se comprend facilement par le fait qu'on pourrait difficilement gérer tous les objets dans la même instance. Dans tous les cas, il est plus simple d'avoir un modèle qui correspond au fonctionnement logique du système,

à savoir qu'un événement relatif à un objet donné fasse effectivement évoluer l'état de cet objet, et de cet objet seulement. Par la suite, cela nous permet d'être plus précis par rapport au diagnostic de la cohérence.

Dans Trace Compass, cette distinction est possible, puisque l'on peut gérer les différentes FSMs dans une seule et même analyse XML, et définir dans chacune les événements adéquats permettant d'identifier la FSM à utiliser.

Un des avantages de l'utilisation des machines à états finis pour représenter les systèmes étudiés est que celles-ci peuvent être facilement reconstruites à partir de représentations utilisant d'autres formalismes (par exemple, on peut construire une FSM à partir d'un réseau de Pétri) (Lunze and Schröder, 2001).

De plus, ce sont des objets qui ont été très étudiés (littérature pertinente en grand nombre, beaucoup de propriétés démontrées) et, contrairement à d'autres formalismes, ils possèdent une définition formelle et bien connue. Ainsi, on a à notre disposition de nombreux opérateurs mathématiques (notamment la composition) très utiles pour diverses applications.

Parmi les nombreux travaux portant sur l'application de ce formalisme de représentation, Matni and Dagenais (2009) ont montré qu'il était possible de réaliser l'analyse de traces avec des machines à états, ce qui permet de vérifier de manière automatique des motifs de comportements anormaux (par exemple, une attaque de type *SYN-flood*) dans une trace donnée. Pour cela, le langage State Machine [21] est utilisé. Cela permet, entre autres, de créer des événements synthétiques, tels qu'un événement d'avertissement.

Tout d'abord, il faut définir le motif. Cela permet de construire la machine à états finis associée, écrite en langage State Machine. Puis, cette FSM devient l'entrée fournie au SMC (*State Machine Compiler*), qui donne en sortie un fichier en C, C++ ou Java. Ce dernier est lié au *checker*, qui est une bibliothèque partagée liée à l'outil de visualisation de traces. Enfin, on peut effectuer la vérification du motif avec l'outil.

Une contribution essentielle à la théorie des FSMs, dont nous aurons besoin dans la suite de notre travail, est l'algorithme, dit algorithme de Dijkstra, permettant de calculer le chemin le plus court entre deux nœuds d'un graphe (Dijkstra, 1959). Dans cet article, deux problèmes sont exposés avec la solution proposée pour chacun. Le premier problème est appelé "*Minimum spanning tree*" et consiste à chercher l'arbre couvrant [22] minimum, c'est-à-dire l'arbre

21. compilateur open-source disponible : http ://smc.sourceforge.net/

22. arbre sous-graphe d'un graphe G non dirigé qui comprend tous les nœuds de G et un nombre minimum d'arêtes (https ://en.wikipedia.org/wiki/Spanning_tree)

couvrant dont la somme des poids des arêtes est minimum. Cela nécessite donc d'avoir un graphe G pondéré. Pour cela, l'arbre est construit arête par arête, en sélectionnant à chaque itération l'arête de poids le plus petit parmi un ensemble de candidats, tant que tous les nœuds n'ont pas été atteints.

Le second problème est celui du chemin le plus court (appelé "*Shortest path (between 2 nodes)*"). Lié au premier problème, cela consiste en une construction incrémentale depuis le nœud de départ jusqu'à atteindre le nœud cible, en utilisant le fait suivant : savoir que R est un nœud sur le chemin minimum entre P et Q implique de connaître le chemin minimum entre P et R.

Il faut noter un travail intéressant concernant la génération de FSMs à partir de traces (Koskimies and Mäkinen, 1994). Dans ce contexte, une trace est représentée par une séquence de paires (e1, e2) où e1, e2 $\in$ F (F étant l'ensemble des événements du système ; e1 l'événement de l'arc sortant (ce qui correspond à "objet envoie un événement"), e2 l'événement de l'arc entrant ("objet reçoit un événement"). Chaque paire représente donc le fait qu'on ait atteint un état qui a une action e1 et une transition étiquetée par e2 (e1 null $\Rightarrow$ pas d'action, e2 null $\Rightarrow$ fin de la trace).

Le principe de la méthode proposée est de prendre en entrée un diagramme de trace, qui est en fait un diagramme de séquence en UML/MSC écrit en SDL. À partir de ce diagramme de trace, on peut générer la trace correspondante, qui va être l'entrée de l'algorithme. En sortie, on obtient une machine à états.

Notons que la définition des traces est un peu primitive. Néanmoins, cela ouvre des perspectives intéressantes quant au fait qu'il serait possible d'automatiser la définition des FSMs à partir des traces d'un système donné. Dans le cas idéal, on aurait un ensemble de traces pour le système, traces que l'on sait être correctes (c'est-à-dire, sans événements manquants, ni incohérences) ; et on construirait un modèle du système tracé à partir de cet ensemble de traces. Autrement, puisque l'on connaît l'intervalle où des événements sont perdus (voir 1.2), et que l'on a accès à la trace avant et après cet intervalle, on pourrait donner la partie de la trace avant l'intervalle perdu comme entrée de l'algorithme, puis utiliser la FSM générée en sortie afin de retrouver les événements perdus.

2.3.2 Implémentation des FSMs pour l'analyse de traces

Les travaux de maîtrise de Wininger et Kouamé (voir Wininger, 2014; Kouamé, 2015) ont porté sur la mise en place d'un langage déclaratif permettant de définir des modèles à états,

puis des machines à états, à partir d'une trace. Nous allons présenter brièvement son fonctionnement.

Comme cela a été expliqué dans la partie 2.2.2, la responsabilité de la définition de la structure du système d'états incombe au développeur, et est propre à chaque analyse. Il est donc possible de définir en XML des gestionnaires d'événements de façon indépendante de Trace Compass (qui analyse les traces).

Dans la solution proposée, un scénario est une instance d'une FSM. Cela signifie qu'il peut y avoir plusieurs "objets FSM" fonctionnant en même temps, chacun étant représenté par un scénario.

La solution fonctionne en deux étapes principales :

1. création du modèle : l'utilisateur définit les patrons avec le langage de description, puis le XML est analysé par le gestionnaire d'événements qui crée les modèles Java correspondants, pour chaque entité nécessaire ;
2. filtrage d'événements : l'analyseur reçoit des événements depuis la trace, son gestionnaire d'événements traite les événements les uns après les autres en passant l'événement à la FSM correspondante, qui le transfère aux scénarios qui traitent l'événement (produit une action, une sortie...). L'analyseur de filtre est responsable de la création des scénarios, qui sont des instances d'un patron modélisé à l'étape précédente.

Les états de la FSM sont accessibles dans l'arbre des attributs, comme pour n'importe quel système d'états dans Trace Compass, ce qui est très utile, car cela signifie que l'on y a accès depuis l'analyse XML.

Ces travaux apportent une plus grande flexibilité à Trace Compass en permettant à l'utilisateur de définir ses propres analyses, dans un langage facile d'utilisation. L'expérience d'utilisateur s'en trouve grandement améliorée. Néanmoins, ils présentent certaines limites, que nous allons exposer.

Déjà, dans le mémoire de Wininger, le problème des événements manquants est identifié.

> "De même, un autre problème lié au fonctionnement par changement d'état est la mauvaise résilience du gestionnaire d'état par rapport à la perte d'événements. Pour la même raison que dans l'initialisation, en cas de perte d'un événement, on reste dans un état incorrect pendant une durée indéterminée."

Si l'on perd un événement qui aurait dû provoquer un changement d'état, l'état actuel n'est pas modifié et tant que l'on n'atteint pas un état dont la cohérence est certaine (nous verrons plus tard comment cela peut être défini), il n'est pas possible de savoir si l'état est valide.

2.3.3 Applications utilisant les FSMs

Les machines à états finis sont très utilisées dans le domaine de l'analyse lexicale et syntaxique. On peut notamment penser aux compilateurs, dont l'analyse lexicale est la première étape, qui produit les symboles utilisés ensuite pour l'analyse syntaxique (Aho et al., 1986). Lors de l'analyse lexicale, des tables de transition (qui implémentent des FSMs) sont utilisées pour reconnaître les caractères en entrée. Finalement, un analyseur lexical utilise des FSMs en leur fournissant des caractères en entrée, et en récupérant les symboles (ou les erreurs, si les chaînes de caractères ne sont pas reconnues) en sortie. L'analyse syntaxique vérifie que les symboles sont acceptés par le langage à partir de la grammaire associée, en produisant un arbre d'analyse syntaxique (*parse tree*). De la même façon que précédemment, un analyseur syntaxique (*parser*) utilise la FSM correspondant à la grammaire.

Nombre des techniques employées pour l'analyse lexicale et syntaxique peuvent être retrouvées dans des utilitaires communs : éditeur de texte, recherche de patrons...

Les FSMs sont aussi utilisées dans le domaine des réseaux, en particulier pour les protocoles de communication. Par exemple, une connexion TCP est définie, dans la spécification du protocole, comme étant dans une succession d'états parmi *LISTEN, SYN-SENT, SYN-RECEIVED, ESTABLISHED, FIN-WAIT-1, FIN-WAIT-2, CLOSE-WAIT, CLOSING, LAST-ACK, TIME-WAIT, CLOSED* (of Southern California, 1981, où un diagramme d'états est spécifié à la figure 6) : c'est une machine à états.

Dans les sections suivantes, nous allons étudier plus en détail les domaines mentionnés, et comment les erreurs y sont traitées.

2.4 Diagnostic d'erreurs

Nous allons maintenant nous pencher sur les différentes techniques de diagnostic d'erreurs, selon leur domaine d'application. Nous verrons quelles sont les innovations qu'elles apportent, mais aussi quelles sont leurs limites.

2.4.1 DES

Un événement peut être défini selon deux propriétés : il se produit instantanément, et il provoque une transition d'une valeur d'état (*state value*) à une autre (Cassandras and Lafortune, 2008).

Les DES (*Discrete Event Systems*) sont également définis par deux propriétés : l'espace des états est un ensemble discret (ensemble fini), et le mécanisme de transition d'états est basé sur les événements (*event-driven*), c'est-à-dire que les événements du système ont lieu à un instant t déterminé, qui ne correspond pas nécessairement à un tick d'horloge (asynchronicité) et l'état ne peut changer qu'à ces instants t précis (ce qui est différent d'un mécanisme basé sur le temps – *time-driven*). Un tel système fait partie de la classe des systèmes dynamiques (la sortie du système dépend des valeurs passées de l'entrée), invariants au temps (le comportement du système ne change pas en fonction du temps), et non linéraires.

D'après ces définitions, un système informatique est donc un DES (Sampath et al., 1996).

On peut facilement faire le lien entre un langage et un DES (Cassandras and Lafortune, 2008). En effet, l'alphabet est l'ensemble (fini) d'événements E. Un mot est alors une séquence d'événements (soit une trace). Il est pertinent d'utiliser une FSM pour traiter des traces, puisqu'elle permet de reconnaître un langage à partir de règles définies, en utilisant des événements (éléments de l'alphabet) comme étiquettes des transitions.

Le diagnostic d'erreur est généralement défini à partir de deux objectifs principaux Zaytoon and Sayed-Mouchaweh (2012) :

— la détection d'erreurs : déterminer si un défaut a eu lieu ou non dans le système ;

— l'isolation d'erreurs : identifier le composant responsable du défaut.

Il peut être réalisé :

— en ligne : le système fonctionne dans des conditions normales et le diagnostic n'est basé que sur les sorties ;

— hors-ligne : le système fonctionne dans un environnement de test (*test-bed*), c'est-à-dire qu'il ne fonctionne pas dans des conditions normales et que le diagnostic est basé sur les entrées et les sorties.

Dans cette recherche, on se trouve dans le cas où le diagnostic est fait en ligne, car on n'a accès qu'à la trace (c'est-à-dire les sorties).

La littérature dans le domaine du diagnostic d'erreurs pour les DES est très vaste (Zaytoon and Lafortune, 2013). On distingue plusieurs formalismes de représentation :

— les automates (FSM notamment)

— les réseaux de Pétri

— les diagrammes d'états (*statecharts*)

Un des principaux défis identifiés est celui du fléau de la dimension (*curse of dimensionality*) qui peut causer une explosion combinatoire des solutions possibles. De plus, il faut être capable de supporter des systèmes variés et complexes : systèmes distribués, systèmes hétérogènes, etc.

Sampath et al. (1996) est un article classique et fondateur du domaine du diagnostic de DES. Les informations collectées sont traitées dans un cadre purement événementiel.

Un concept important qui y est défini est celui de la I-diagnosticabilité (*I-diagnosability*). Cela correspond au fait que tout cas d'erreur peut être trouvé avec certitude après un certain nombre fini d'événements, à partir de l'occurrence d'un événement indicateur (*indicator event*).

L'idée des recherches sur le diagnostic d'erreurs pour les DES repose sur la définition d'un automate A dont certaines actions ne sont pas observables : l'ensemble des actions possibles est divisé entre un sous-ensemble d'actions observables et un sous-ensemble d'actions non observables, souvent assimilées à des erreurs. L'objectif est de déterminer s'il y a pu avoir une erreur (non observée) parmi une séquence d'actions (un mot ω obtenu par une exécution de A) (Fabre and Jezequel, 2010).

Pour ce faire, deux automates, assez similaires, peuvent être construits :

— l'observateur (*observer*) : c'est une paire formée d'une FSM déterministe sur l'alphabet des actions observables, et d'une fonction d'étiquetage sur les états de A. La sortie est un ensemble des états possibles, dans lequel A aurait pu être, suite à l'observation du mot ω ;

— le diagnostiqueur (*diagnoser*) : il est similaire à un observateur, à l'exception de la fonction d'étiquetage qui retourne "faute" si parmi tous les chemins qu'auraient pu produire le mot, au moins un contient une transition avec une erreur, ou "valide" si aucun chemin ne contient une transition avec une erreur.

On construit l'observateur par une ε-réduction sur l'automate A (transition des transitions étiquetées par des événements non observables), suivie par une transformation de l'automate non déterministe en son équivalent déterministe par l'union d'états (*determinization*) :

$$Obs(A) = Det(Red(A))$$

Finalement, le diagnostiqueur est un observateur pour l'automate A, augmenté par une mé-

moire des types de transitions rencontrés sur le chemin. On utilise la même méthode de construction sur l'automate augmenté, une fois que celui-ci a été construit (Fabre, 2013).

Les concepts de base ont été développés pour un modèle global, mais de nombreuses recherches ont porté sur l'application de ces méthodes pour des cas distribués (Sayed-Mouchaweh, 2012). On a alors un ensemble de modèles locaux.

On peut aussi considérer les cas où le diagnostiqueur est probabiliste. Dans ce cas, une probabilité est associée à chaque transition et on calcule quel est le chemin emprunté avec la plus forte probabilité.

Quel que soit le système considéré, ces méthodes utilisent au moins un modèle sans faute du système. Nous pouvons utiliser cela à notre avantage, puisque l'on a déjà le modèle à notre disposition (la FSM, qui est idéalement fournie par un expert du système connaissant bien son fonctionnement normal).

Malgré cela, ces méthodes sont difficilement applicables à notre cas d'étude, car nous n'avons pas d'événement provoquant une faute à proprement parler. On pourrait considérer les événements perdus comme des événements non observables, mais tout événement peut potentiellement être perdu, donc notre système n'est pas diagnosticable.

En effet, cela crée la présence de cycles d'événements non observables et de cycles d'événements observables dans notre système, ce qui cause une incertitude sur l'occurrence d'une erreur (*(Fi, Ii)-uncertain state*). Par exemple, l'état d'un processus peut sembler cohérent alors qu'il y a eu un cycle d'événements non observables, car on est retourné à un état cohérent au sein d'un cycle d'événements observables.

Plus formellement, on peut prouver la non diagnosticabilité par la méthode de la machine jumelée (*twin machine*) – voir en annexes, la figure A.1.

2.4.2 Grammaires

Les grammaires sont très souvent étudiées en lien avec les automates. C'est pourquoi il est pertinent de s'intéresser à la façon dont les erreurs peuvent être diagnostiquées à partir des travaux sur les grammaires, notamment dans le contexte des compilateurs, car les grammaires non contextuelles sont très utilisées pour décrire la syntaxe des langages de programmation (Earley, 1970).

Une grammaire non contextuelle est une grammaire où les règles de production sont de la forme : $X \rightarrow \alpha$; où X est un symbole non terminal, α une chaîne de symboles terminaux ou

non terminaux.

On comprend l'appellation "non contextuelle" par le fait que ce type de règle de production ne prend pas en compte le contexte d'occurrence du symbole X.

Dans le domaine de la théorie des langages, un article fondateur est celui de Earley où l'auteur présente ce qui est maintenant connu comme l'algorithme d'Earley. Celui-ci décrit comment construire un analyseur syntaxique (programme qui détermine si une chaîne de caractères donnée en entrée est acceptée ou rejetée par le langage, et qui reconnaît l'ensemble des dérivations pour cette chaîne). Celui-ci a l'avantage de supporter les grammaires non contextuelles, quelle que soit leur forme.

Le diagnostic d'erreurs est assez simple à effectuer avec les grammaires : il suffit de construire un analyseur syntaxique à partir de la grammaire considérée, et de fournir les chaînes de caractères à vérifier en entrée ; si l'analyseur accepte la chaîne, elle est correcte, sinon elle contient une erreur à l'endroit où l'analyse se termine. Nous renvoyons à la sous-section 2.5.3 pour plus de détails sur la correction de ces erreurs, qui constitue le cœur de la recherche dans ce domaine.

La simple détection d'erreurs syntaxiques ou lexicales (sans correction) est de peu d'utilité, outre la validation d'un document. Nous n'avons pas vraiment d'intérêt à vérifier la validité d'une trace, puisqu'on suppose qu'il ne peut pas y avoir d'insertion ou de substitution d'événements.

2.4.3 Extraction de processus métiers

Le domaine de la gestion de processus métiers (*business process management*), bien que non directement relié à l'informatique, est à l'origine de beaucoup de travaux sur le diagnostic et la correction d'erreurs à partir de modèles, qui sont généralisables à d'autres cas où des modèles similaires sont utilisés.

Des concepts importants sont introduits dans Rozinat and van der Aalst (2008), où l'on trouve notamment une définition de l'extraction de processus métiers. Ses trois étapes constitutives sont identifiées :

— la découverte : trouver le modèle du processus à partir d'un journal (*log*) ;
— la vérification de la conformité (*conformance checking*) : comparer le modèle et un journal ;
— l'extension : diagnostiquer un processus à partir d'un journal projeté sur le modèle pour

améliorer ce dernier.

La lecture d'une trace événement par événement, lors d'une analyse, correspond à ce qui est appelé la réexécution (*replay*). La conformité est définie comme la réponse à la question «Existe-t-il une bonne correspondance entre les événements enregistrés et le modèle ? ». Dans notre cas, on n'a pas à se soucier de problèmes dans le journal, car on sait (en tous cas, l'hypothèse est faible) que tous les événements dans la trace ont effectivement eu lieu sur le système (on ne peut pas avoir d'activité supplémentaire qui apparaît dans le journal sans avoir eu lieu sur le système).

Verbeek et al. (2001) proposent un outil pour détecter et corriger des erreurs dans un processus de flux de production (*workflow process*). Ils utilisent pour cela des réseaux de Pétri. La définition d'un processus indique quelles tâches doivent être exécutées, et dans quel ordre, ce que l'on peut mettre en parallèle avec le modèle d'un système. De même, le processus est exécuté pour un cas particulier, ce qui pourrait correspondre avec une trace. On voit donc des similarités avec le contexte de ce travail, même si nous utilisons des FSMs et non pas des réseaux de Pétri, dans notre cas. La même approche peut être trouvée dans Masellis et al. (2017).

Comme mentionné plus haut, une des problématiques importantes dans ce domaine est la vérification de la conformité d'un journal d'événements, d'une trace, c'est-à-dire l'exécution réelle d'un processus, avec le modèle défini pour ce processus. Dans Adriansyah et al. (2011), les auteurs présentent une technique de réexécution permettant de quantifier la conformité d'un journal donné. Pour juger de la conformité, la métrique mesurée est la *fitness*, qui est calculée de telle sorte qu'elle prend en compte l'importance relative de chaque déviation du modèle, grâce à des fonctions de coût pour chaque type de déviation (ici, une insertion ou une suppression). Ces fonctions peuvent être paramétrées par l'utilisateur. L'article propose donc également le moyen de mesurer cette métrique.

L'objectif est de trouver la meilleure instance du modèle, c'est-à-dire celle qui permet d'obtenir la meilleure valeur de *fitness* pour un coût minimal (des déviations), pour un journal d'événements donné. Cela peut être utilisé pour indiquer à l'utilisateur quels événements (on parle plutôt d'activités, dans ce contexte) ont été perdus ou ajoutés par erreur. Les instances sont construites au cours de la réexécution du journal sur le modèle, et la meilleure est sélectionnée grâce à l'algorithme A*, dont le domaine est associé à l'espace de recherche. À noter que, dans notre cas, on n'a pas à se soucier d'une déviation du type insertion, puisqu'un événement n'ayant pas eu lieu dans le système ne peut pas avoir été écrit dans la trace considérée.

De Leoni et al. (2012) s'intéressent au cas des modèles déclaratifs, qui spécifient seulement les contraintes, contrairement aux modèles procéduraux qui spécifient seulement le comportement autorisé. Un des défis est de ne pas explorer l'ensemble de l'espace de recherche, qui est bien plus grand. Pour chaque contrainte décrite dans le modèle, une machine à états finis, appelée automate de contrainte (*Constraint Automaton*), est créée. Ces FSM sont utilisées pour vérifier que le journal respecte chaque contrainte. Même si l'application des modèles déclaratifs diffèrent de la nôtre, la technique d'alignement proposée est intéressante.

L'alignement consiste à superposer chaque événement à une activité du modèle. Le problème de trouver un alignement optimal correspond donc au fait de chercher, dans le modèle du processus considéré, le chemin le plus proche de la séquence d'événements du journal (notons que ce problème est NP-difficile). En associant une fonction de coût au modèle, les auteurs utilisent l'algorithme A* pour trouver un alignement optimal. Ensuite, les auteurs se basent sur la *fitness* pour évaluer la conformité du journal par rapport au modèle, en définissant une fonction de *fitness* qui prend en compte le coût de l'alignement optimal.

Il y a des améliorations de cette technique. Par exemple, Song et al. (2017) se basent sur les travaux précédemment mentionnés et d'anciens travaux (Song et al., 2015) pour proposer une méthode d'alignement dont les performances permettent de traiter des journaux d'événements de taille industrielle réelle, dans la plupart des cas. Cette méthode utilise l'élagage (*pruning*) à partir d'heuristiques, afin de restreindre l'espace de recherche en supprimant les branches ne pouvant pas mener à la solution optimale.

2.4.4 Protocole réseau

La perte de paquets sur un réseau est une problématique très étudiée. Toutefois, l'application pratique de la simple détection des erreurs est assez limitée, c'est pourquoi il faudra se rapporter à la sous-section 2.5.2 pour une plus abondante littérature sur le sujet de la correction de ce problème.

Ce phénomène peut s'expliquer de plusieurs façons : canal de communication peu fiable, bruit sur le canal, saturation du réseau due à une faible bande passante... Certains cas sont assez similaires à ceux que l'on observe lors de la perte d'événements, c'est pourquoi il est pertinent d'étudier ce domaine.

On peut commencer par mentionner l'article fondateur Hamming qui introduit la distance de Hamming. Il peut être appliqué au domaine des télécommunications où l'information

transmise peut être un signal binaire facilement utilisable avec cette méthode.

Un des codes de détection d'erreur le plus simple est le contrôle de parité : on ajoute un bit, appelé bit de parité, qui indique s'il le nombre de 1 dans le message est pair ou non. Cela permet de détecter le cas où il y a une seule erreur de transmission, ce qui modifierait la propriété paire ou impaire protégée par le bit de parité (et on voit bien que ce n'est pas nécessairement vrai s'il y a deux erreurs, ou plus).

Une autre technique importante est celle des codes cycliques appliqués à la détection d'erreurs, technique introduite dans Peterson and Brown (1961). Elle exploite les principes suivants. L'information à transmettre est codée sur la forme d'un message binaire de taille n, composé de k bits de contenu suivis de n-k bits de contrôle. Ce message peut être représenté sous forme polynomiale. On définit un code polynomial comme un polynôme (de degré inférieur à n) divisible par un polynôme (de degré n-k) donné, appelé générateur polynomial. Si l'on code le message à envoyer avec le générateur polynomial, on peut alors vérifier la présence d'erreurs à la réception : si le code reçu n'est pas divisible par le générateur, cela signifie qu'il y a au moins une erreur dans le message reçu (mais la réciproque n'est pas vraie). Les bits de contrôle correspondent donc au reste de la division polynomiale des bits de contenu avec le générateur.

Cette technique donne la base d'un ensemble de méthodes, que l'on appelle le contrôle par redondance cyclique (*cyclic redundancy check*). Elle est particulièrement appropriée pour des données binaires transmises sur un réseau (signaux digitaux).

On peut utiliser le contrôle par redondance cyclique pour générer des sommes de contrôle (*checksums*). Celles-ci permettent de vérifier l'intégrité des données transmises. On peut, par exemple, citer les mécanismes de contrôle d'erreurs de la suite de protocoles de communication TCP/IP, pour lesquels les sommes de contrôle sont utilisées aux différentes couches. Ainsi, un datagramme IP contient un champ pour la somme de contrôle de l'en-tête (Braden, 1989) ; de même qu'un paquet TCP contient aussi une somme de contrôle qui protège son contenu (of Southern California, 1981). CRC-32, un des principaux algorithmes de contrôle par redondance cyclique, est utilisé dans la plupart des LANs (Halsall, 2005).

Le format de représentation des données utilisé par ces applications est assez éloigné du nôtre. Il repose sur des propriétés mathématiques qui s'appliquent bien aux signaux, mais pas à une suite d'événements dans une trace.

2.5 Correction d'erreurs

Après avoir détecté, voire diagnostiqué, une erreur, l'étape suivante consiste à la corriger. Encore une fois, selon le domaine d'application, différentes solutions sont proposées. Nous allons nous y intéresser, et identifier les limites de chacune.

On peut d'abord mentionner rapidement l'article de Gladyshev and Patel, qui formalise le problème de la reconstruction d'états manquants avec une FSM, appliqué au domaine de l'investigation numérique. Une des limitations identifiées dans l'article est le fait qu'il faudrait prendre en compte des attributs supplémentaires du système, comme des statistiques. C'est ce que nous nous proposons de faire dans notre solution, par exemple en prenant en compte les probabilités d'occurrence des événements pour chaque scénario.

2.5.1 En statistiques

De manière classique, on définit une variable R qui contient des valeurs binaires (1 si la variable X est observée, 0 si la valeur est manquante pour X) afin de caractériser les données manquantes dans une variable X. La perte de valeurs est alors traitée comme un phénomène probabiliste (avec R un ensemble de valeurs aléatoires) (Schafer and Graham, 2002).

En statistique, on peut classer les données manquantes selon le mécanisme qui a conduit à leur perte (Allison, 2012). Ce dernier peut être :

— MAR (*Missing At Random*) : la probabilité qu'il manque une observation pour une variable aléatoire X peut dépendre d'une autre variable observable, mais pas de X elle-même ;

— MCAR (*Missing Completely At Random*) : la probabilité qu'il manque une observation pour une variable aléatoire X ne dépend d'aucune autre variable observable, ni de X (c'est donc un cas particulier de MAR) ;

— NMAR (*Not Missing At Random*) : le mécanisme n'est pas MAR.

On peut aussi classifier les données selon le type d'absence de réponse (Schafer and Graham, 2002) :

— absence de réponse de l'entité (*unit nonresponse*) : toute la procédure de collecte de données échoue pour une entité donnée, car cette entité chargée de donner des réponses ne peut/veut pas répondre ;

— absence de réponse de l'élément (*item nonresponse*) : la collecte de données sur certaines variables seulement échoue (on a des données partielles pour une entité donnée).

La technique à appliquer pour supporter des données manquantes dépend de la façon dont ces données le sont devenues (García-Laencina et al., 2010). Lorsque le mécanisme est de type MCAR ou MAR, on dit que les données manquantes peuvent "être ignorées".

Une méthode pour traiter le type MAR devrait se conformer aux exigences suivantes (Allison, 2012) :

— minimiser le biais introduit par l'application de la méthode ;

— utiliser le plus possible les informations disponibles ;

— produire une bonne évaluation de l'incertitude ;

— (éventuellement) ne pas faire d'hypothèses trop restrictives sur le mécanisme des données manquantes.

Ce sont des règles importantes à garder à l'esprit tout au long de notre travail.

Il y a alors plusieurs méthodes pour gérer les données manquantes (García-Laencina et al., 2010) :

— suppression des cas où il y a des données manquantes ;

— imputation statistique ;

— modélisation basée sur la distribution des données d'entrée.

Dans notre cas, nous ne sommes pas dans le domaine continu, puisque nos données sont ponctuelles (ce sont des événements). De plus, il faut noter que l'on a un moyen de retrouver (même si cela n'est que partiellement) les données manquantes à partir des FSMs. Dans la plupart des cas classiques de la littérature en statistiques, les données manquantes n'existent tout simplement pas (pas de réponse d'un sujet dans un questionnaire, par exemple).

On peut aussi ajouter que l'on ne fait pas vraiment de statistiques sur nos données, juste de l'observation et de l'analyse. Il n'y a donc pas de méthode statistique à appliquer, et il n'est alors pas important d'avoir des échantillons statistiques bien faits. On considère plutôt des méthodes heuristiques.

2.5.2 Codes correcteurs

Avant de nous intéresser plus particulièrement aux codes correcteurs, nous pouvons voir quelles sont les différentes manières de gérer les pertes de paquets dans le domaine de la transmission de signaux. Perkins et al. s'intéresse aux méthodes permettant de pallier les pertes de paquet dans un contexte de multidiffusion IP (*IP multicasting*) pour les applications multimédia. Les auteurs identifient deux catégories de méthodes de récupération de paquets, selon que celles-ci se basent :

— sur l'émetteur. Elles sont elles-mêmes séparées en deux catégories :

- correction active : l'émetteur retransmet les paquets perdus. C'est une technique assez directe de réparation d'erreurs, mais elle n'est pas applicable si un grand nombre de paquets sont perdus, ou si l'application doit respecter une faible valeur de latence. Citons, par exemple, la méthode dite de demande automatique de répétition (*Automatic repeat request – ARQ*).
- correction passive : l'information transmise contient des éléments permettant d'appliquer des corrections, si nécessaire. Il existe principalement deux méthodes : l'entrelacement (*interleaving*), qui consiste à réorganiser les données à transmettre de manière à mitiger l'effet d'une erreur (puisque cela va disperser l'erreur à plusieurs endroits du message, une fois les données replacées dans le bon ordre après réception), et le contrôle continu (*forward error correction*), qui consiste à ajouter des données aux paquets (redondance). La première méthode ne nécessite pas plus de bande passante, mais augmente la latence. La seconde méthode, en augmentant la quantité de données à transmettre, nécessite plus de bande passante.

— sur le destinataire. Ces méthodes sont en général utilisées lorsqu'il n'a pas été possible de corriger les erreurs avec les méthodes précédentes basées sur l'émetteur, soit à cause d'un échec, soit à cause d'une impossibilité technique (par exemple, un canal de communication à sens unique). On parle alors de dissimulation des erreurs (*error concealment*), et on les divise en trois catégories :

- insertion : on insère un paquet de "remplissage" (répétition du paquet précédent, bruit, etc.) ;
- interpolation : on reconstruit un paquet par interpolation, à partir des autres paquets ;
- regénération : on construit un paquet à partir des informations du décodeur (paramètres, algorithme utilisé...) et des paquets précédents et suivants.

Comme on peut le supposer, l'essentiel des recherches dans ce domaine vont donc porter sur les codes correcteurs d'erreur. Notamment, une méthode importante est celle de l'algorithme de Viterbi (Viterbi, 1967; Ryan and Nudd, 1993). L'objectif est de trouver le chemin le plus probable dans un treillis, en fonction d'un ensemble d'observations donné. Le treillis peut être construit à partir d'une FSM, ce qui est intéressant dans notre cas. Toutefois, nous n'avons pas un HMM (*Hidden Markov Model*) car on connaît la FSM, elle est disponible.

L'algorithme fonctionne de la manière suivante : à partir de chaque état t-1, on teste chaque transition possible en calculant la différence entre l'observation actuelle et la sortie attendue, ce qui donne le poids de la transition. Puis on choisit, pour chaque état t, une des transitions

avec le poids le plus faible. Cette méthode supporte la correction de données suite à un changement de symboles.

L'algorithm est appliqué dans Bouloutas et al. (1991). Les auteurs cherchent à répondre à la problématique de l'ECP (*Error-Correcting Parsing*), ce qui consiste à trouver le meilleur chemin dans un treillis. En plus de cette application, ils proposent une extension de l'algorithme aux cas où il y aurait non seulement une insertion, mais aussi une substitution ou une suppression dans le modèle, ainsi que pour les cas où il y aurait des cycles dans le treillis.

Pour cela, on construit un estimateur qui va réaliser le symétrique des opérations faites sur la sortie de la FSM (si un symbole a été supprimé, l'estimateur va ajouter un symbole). Il va alors générer la meilleure estimation de la sortie originale, ainsi que la meilleure séquence d'opérations qui ont été effectuées sur la sortie modifiée pour obtenir l'estimation. On obtient également le chemin dans la FSM qui mène à cette estimation. Le coût des opérations est additif.

Il y a deux cas possibles :

— les données sont sous la forme d'une chaîne de caractères : on combine (produit) la FSM générant les données originales G et la FSM des données reçues D (qui nous donne une chaîne d'états), puis on recherche le chemin de coût minimal depuis l'ensemble des états initiaux vers un état final ;

— les données sont sous la forme d'un ensemble de chaînes de caractères, c'est-à-dire d'un langage régulier représenté par une FSM (généralisation du premier cas), ce qui correspond au cas où seule une observation partielle de la séquence reçue est possible (donc on sait juste qu'elle fait partie de l'ensemble de séquences qui peuvent être générées par la FSM des données D) : on construit le produit de la même façon que dans le premier cas, mais on cherche la séquence (chemin) générée par la FSM G de coût minimal par rapport à l'ensemble des séquences générées par la FSM D. Néanmoins, on peut appliquer le même algorithme pour les deux cas.

L'application principale se trouve au niveau des protocoles de communication, pour les cas où l'observation complète des événements n'est pas possible (par exemple, un canal de communication est désactivé pour une durée donnée ; ou encore, une irruption soudaine du bruit sur les canaux apparaît).

Dans Amengual and Vidal (1998), l'algorithme de Viterbi est appliqué au domaine de l'analyse syntaxique (reconnaissance de motifs) afin de corriger des erreurs durant l'analyse. Les auteurs adaptent la solution proposée par Bouloutas et al. (1991) en mettant l'accent sur la performance de leur méthode.

Néanmoins, l'algorithme de Viterbi ne s'applique pas à notre cas, du fait d'un contexte différent. En effet, nous avons accès à plus d'informations dans les événements pour déduire des choses, par rapport à un signal simple comme ceux utilisés dans l'algorithme de Viterbi. Nous pouvons (et devons) donc tirer parti de ces informations. De plus, la redondance propre aux signaux numériques ne peut pas être exploitée pour une trace, qui ne possède pas les mêmes caractéristiques.

2.5.3 Correction d'erreurs lexicales et syntaxiques

Graham and Rhodes (1975) présentent un état de l'art du domaine qu'il est intéressant de reproduire ici. Les différentes catégories de méthodes sont divisées comme suit :

— la spécification manuelle des erreurs les plus courantes. Cela requiert des efforts supplémentaires importants. De plus, il y a un problème si une erreur imprévue a lieu ;

— le "mode panique" (*"panic mode"*), consistant à lire l'entrée jusqu'à un symbole d'une classe spéciale (par exemple, un ' ;') qui provoque alors le dépilement de la pile d'analyse syntaxique jusqu'à ce que le symbole spécial puisse en être la suite. C'est une des techniques les plus simples, mais il y a le risque de perdre d'éventuelles erreurs dans la partie du code qui est dépilée, et cela ne fournit aucune information sur la nature de l'erreur ;

— la technique de la distance minimale, consistant à déterminer le nombre minimum d'opérations d'édition pour transformer la chaîne erronée en une chaîne valide. Parmi les limitations, cette méthode ne s'exécute pas en un temps linéaire alors que c'est le cas de l'analyse syntaxique. De plus, elle ne trouve pas forcément la meilleure solution. En fait, des travaux ultérieurs vont permettre de grandes avancées dans ce domaine (par exemple, voir plus loin Boullier and Jourdan, 1987). Ce type de correction d'erreurs peut être divisé en deux catégories (Aho et al., 1986) : correction locale, et correction globale.

Rappelons qu'il y a trois types d'erreurs syntaxiques (Aho and Peterson, 1972) :

— remplacement d'un symbole ;

— suppression d'un symbole ;

— insertion d'un symbole.

Le problème que les auteurs cherchent à résoudre dans Aho and Peterson (1972) est énoncé comme la recherche d'un algorithme pouvant trouver un arbre d'analyse syntaxique pour une chaîne de caractères w dans L(G) dont la distance à x est minimale, étant donnée une grammaire G dont l'alphabet est Σ et x une chaîne de caractères dans Σ^*.

La méthode proposée consiste en l'ajout d'un ensemble de règles de production pour les erreurs afin d'étendre la grammaire (ce qui nous donne une grammaire couvrante – *covering grammar*), tel que L(G') = Σ^*. Cette grammaire est ensuite utilisée par un parseur qui utilise le moins de règles de production d'erreurs possible ; là où une règle d'erreur est utilisée, il y a une erreur.

Dans notre cas, on ne veut pas avoir à spécifier les erreurs, car il existe une infinité d'erreurs possibles. De plus, la complexité est très grande : $O(n^3)$, où n = |chaîne de caractères en entrée|. Notons que les FSMs (langage régulier) sont une sous-catégorie des grammaires sans contexte, donc on pourrait construire les règles de production nécessaires à cette méthode.

Une méthode un peu plus avancée est proposée dans Graham and Rhodes (1975) : on ne modifie pas ce qui a déjà été scanné, mais on peut modifier le reste de la chaîne pas encore parsée. Avant la phase de correction a lieu une phase de "condensation" : après avoir détecté l'erreur, on tente de réduire la pile autour de l'erreur ('*forward move*' + '*backward move*') ; l'objectif est de faire un résumé du contexte entourant l'erreur, dans le but d'avoir plus d'informations pour traiter l'erreur. On utilise également une distance minimum pondérée, avec des heuristiques pour fixer les coûts.

Il existe de nombreuses extensions des algorithmes précédents, par exemple avec la définition d'un modèle stochastique des erreurs (Lu and Fu, 1977), avec des patrons de correction qui permettent d'exprimer les erreurs courantes et améliorer la qualité des réparations (Yun et al., 1993), ou encore avec l'apport de grammaires probabilistes (Thompson, 1976).

Anderson et al. (1983) proposent une autre technique (appelée *Least-Cost Error Recovery*) pour réparer des erreurs syntaxiques en éditant la chaîne de caractères d'entrée à l'endroit où l'erreur est détectée. Le choix de l'édition dépend des coûts associés aux opérations d'édition, et est effectué par le programmeur. Cette méthode ne prend pas en compte le contexte de l'erreur (notamment, les symboles suivants, qui pourraient donner des informations sur la nature de l'erreur). De plus, les corrections sont appliquées au symbole où l'erreur est détectée, alors qu'une erreur peut être détectée après son occurrence réelle (par exemple, l'erreur pourrait provenir d'un mauvais symbole précédemment).

On peut considérer à la fois la correction locale et celle globale. Boullier and Jourdan (1987) montrent comment l'on peut traiter une erreur dans une source texte. Leur solution est divisée en plusieurs étapes : détection, affichage, puis réparation et récupération. Nous nous

intéressons particulièrement à la dernière étape, qui est elle-même divisée en deux sous-étapes : correction locale et, si c'est un échec, récupération globale, qui correspond au fait que l'erreur ne peut pas être corrigée, mais que l'analyse doit reprendre dans un état correct.

Dans le cas de la correction locale, des portions de texte suffixes du texte avant le point où l'erreur est détectée sont comparées à la source, ainsi qu'à des modèles de correction. Ces modèles sont fournis par le créateur de la grammaire, et spécifient des suppressions, des insertions et/ou des remplacements. S'il y a correspondance, la portion erronée est remplacée par la séquence produite, puis on reprend l'analyse.

Dans le cas de la récupération globale, on lit, sans l'analyser, le texte jusqu'à atteindre un symbole terminal, puis on examine la pile d'analyse grammaticale (*parse stack*). Si on trouve un état avec une transition d'un symbole non terminal suivi d'un symbole terminal, on dépile l'élément de la pile sur cet état, on effectue la transition et on reprend l'analyse. Sinon, on lit le prochain symbole terminal et on recommence.

Dain (1994) construit sur les travaux précédents et propose de générer des préfixes (de longueur δ fixée) pour la chaîne de caractères suivante, plutôt que de calculer toutes les chaînes possibles ; puis de comparer les préfixes à un nombre τ fixé de symboles restants. La solution utilise la distance minimum comme critère de choix, et se base sur le fait que la sous-chaîne située avant une erreur est un préfixe d'une chaîne reconnue par l'analyseur syntaxique, et qu'on peut donc chercher son suffixe le plus probable, qui est aussi le préfixe de la chaîne suivante. C'est donc une technique par anticipation (*lookahead).* On peut faire varier δ ou τ pour améliorer la précision, mais il n'est pas possible de trouver une solution certaine.

2.5.4 Reconstruction du déroulement des processus

Casavanf (1991) étudie le problème de la transformation d'une trace dite corrompue en une trace correcte, et utilise pour cela un modèle du système défini par un réseau de Pétri temporisé. Une trace est corrompue lorsque l'ordre d'exécution des événements a été perturbé par des techniques intrusives de surveillance du système ; cela correspondait à une problématique réelle à l'époque, mais il est maintenant possible de réduire le surcoût avec le traçage, par exemple en utilisant LTTng qui est non bloquant et a un faible surcoût.

Il faut noter que la solution proposée modifie le code source du programme pour ajouter des instructions, invisibles pour l'utilisateur, mais utilisées par la suite pour calculer les corrections. Le réseau de Pétri est calculé automatiquement à partir du code source modifié du programme étudié. La trace corrompue est obtenue après exécution du programme instru-

menté, puis comparée au réseau de Pétri temporisé. Cette dernière opération est réalisée par un mappage de la trace corrompue sur une trace non corrompue par une fonction f indexée par un vecteur de paramètres temporels dépendant de l'exécution.

La solution est assez limitée, puisqu'on n'a pas nécessairement accès au code source de chaque programme. On ne veut pas non plus construire de modèle pour chaque programme.

La problématique étudiée par Rogge-Solti et al. (2013) est celle des événements manquants dans les journaux d'événements documentant un processus, ce qui peut arriver fréquemment quand les journaux sont écrits manuellement (oubli d'un événement réalisé, mauvais événement ajouté...). L'objectif est donc de réparer ces journaux d'événements.

La solution probabiliste proposée fournit une information (supplémentaire par rapport aux autres méthodes) importante en donnant les estampilles temporelles les plus probables pour les événements manquants. Pour cela, des réseaux de Pétri stochastiques sont utilisés comme modèle ; on compare le journal incomplet au modèle par la technique de l'alignement, ce qui produit plusieurs chemins possibles. Le chemin le plus probable est choisi (modèle avec transitions pondérées, prise en compte de la probabilité de perte d'un événement donné, etc.), et on obtient alors un journal dont les événements manquants ont été complétés. La seconde étape consiste à déduire une estampille temporelle pour chaque événement retrouvé.

Toutefois, un chemin dans le modèle ne peut être choisi pour réparation que si au moins un des événements du chemin est effectivement dans le journal incomplet. Cela limite l'application de la solution proposée. De plus, celle-ci nécessite de lire entièrement le journal plusieurs fois (alignement, réexécution pour calculer les probabilités, traitement des estampilles), ce qui serait extrêmement coûteux pour une grande trace de plusieurs millions d'événements.

La solution proposée par Wang et al. (2013) pour récupérer des événements manquants s'applique au domaine de la spécification de processus, définie comme la description des tâches qu'un acteur doit effectuer afin de compléter un objectif métier.

Un processus est ici représenté comme un réseau de Pétri avec une position de départ unique b_{start} et une position de fin unique b_{end}, chaque nœud étant sur un chemin entre b_{start} et b_{end}. Dans ce modèle, une transition est donc un événement dans l'exécution du processus, et une séquence est définie comme une séquence finie d'événements, ce qui correspond à une exécution du processus défini par le réseau de Pétri.

Les techniques existantes consistant à énumérer et rechercher parmi toutes les séquences possibles d'événements sont inefficaces. C'est pourquoi les auteurs proposent de définir la

récupération (*recovery*) d'une séquence o avec un réseau de Pétri N comme la recherche d'une séquence o' telle que o' est définie par N et o est une sous-séquence de o'. La distance entre o et o' est |o'| - |o|.

Ils posent l'hypothèse suivante : choisir la séquence impliquant le moins de changements par rapport à la séquence originale correspond à la récupération optimale, ce qui revient à vouloir minimiser la distance entre o et o'. Toutefois, il faut noter que le problème de la récupération optimale d'événements manquants est NP-complet.

L'algorithme proposé est un algorithme de retour-arrière, dont le rôle est de remplir l'intervalle entre une séquence o et une transition e.

Afin d'améliorer l'efficacité de la solution, une méthode d'embranchement (*branching*) est introduite, se basant sur l'idée qu'il est inutile d'essayer toutes les branches, puisque certaines branches ne vont pas générer une séquence contenant l'événement dont on veut compléter l'intervalle. La méthode consiste donc en la construction d'un index sur les branches afin d'identifier les branches potentiellement valides, et retourne la séquence minimale. Les auteurs envisagent de compléter la solution par des fonctions d'apprentissage pour classer les récupérations et améliorer la précision.

2.6 Conclusion de la revue de littérature

Comme cette revue de littérature a permis de le démontrer, la problématique des événements perdus a été largement étudiée dans le domaine des statistiques (plus particulièrement sous l'angle des données manquantes), de l'analyse lexicale et syntaxique (grammaires), des protocoles, ainsi que des processus métier. Même si la notion d'événement perdu n'est pas toujours la même entre ces différents domaines, il existe de nombreuses méthodes pour détecter et corriger des erreurs dans chacun de ces contextes.

Néanmoins, peu de recherches ont été menées dans le contexte du traçage ; or, une trace contient des informations très précises qui peuvent être exploitées dans le but de traiter ces données manquantes. À partir des FSMs que l'on peut définir facilement en XML dans Trace Compass et d'une trace, il est possible d'étudier l'exécution d'un système et de tirer parti de toutes les informations se trouvant dans chaque événement.

Les problèmes étudiés dans les différents domaines mentionnés sont semblables à celui que nous nous proposons de résoudre. Toutefois, le contexte du traçage est très spécifique, car une trace est très structurée. Ce n'est pas le cas, par exemple, d'un programme analysé par un compilateur, qui ne peut pas se baser sur une structure aussi bien définie. Nous pouvons exploiter cela pour proposer une solution s'appliquant bien au cas particulier des traces.

CHAPITRE 3 MÉTHODOLOGIE

3.1 Tâches à réaliser

Afin de répondre aux objectifs énoncés dans la section 1.3, nous avons identifié plusieurs pistes de solution à approfondir.

L'analyseur fonctionne de façon classique comme suit : dans chaque analyse est défini un fournisseur d'états qui transforme un événement en un (ou plusieurs) changement d'états (quelle que soit la cohérence de ce changement d'états), qui est ensuite stocké sous forme d'un attribut dans une structure de données appelée le système d'états. Une fois l'analyse terminée, les attributs du système d'états sont récupérés pour alimenter une vue, affichée dans l'interface graphique. Il faut donc **contrôler la cohérence de chaque changement d'état**, en fonction du modèle qui a préalablement été fourni. Si un état incohérent est reconnu, il doit être stocké dans une structure particulière du système d'état.

Il faut donc **définir la notion de cohérence**. Du point de vue de la machine à état, la disparition de certains événements implique que des transitions, qui auraient dû être prises, ne le sont pas, puisque les événements qui auraient dû les provoquer ont été perdus. Le nouvel état atteint ne devrait normalement pas être accessible depuis l'état courant, donc l'état courant devient incohérent. À partir de cette définition, il faut **formuler un algorithme pour vérifier la cohérence de chaque événement** selon la définition. Il faut tout d'abord modéliser le système étudié avec une machine à états. Cela consiste à produire un modèle à partir du fonctionnement connu du système, puis l'écrire selon le format défini en XML. Une fois la machine à états produite, il faut l'intégrer à l'algorithme proposé, et **appliquer l'ensemble à une trace de test**. Cette trace correspond à une exécution réelle du système, à laquelle quelques événements ont été sélectionnés et soustraits.

Ensuite, il faut **développer un algorithme permettant de parcourir la machine à états depuis un état incohérent jusqu'au dernier état cohérent le précédant**. À partir de ce parcours et du chemin qui en résulte, un ensemble d'événements peut être déduit. Lorsque plusieurs événements sont possibles pour une même transition, un choix basé sur les probabilités d'occurrence des événements est effectué, ce qui permet d'obtenir la séquence d'événements manquants la plus probable. Les résultats obtenus par l'algorithme sont comparés aux événements préalablement supprimés de la trace. La méthode est validée si les résultats sont similaires, ce qui signifie que les événements retrouvés sont les mêmes que ceux

qui se sont effectivement produits dans le système.

Il faut **donner la liste des événements déduits à l'utilisateur**. Il faut aussi **afficher une vue** avec les événements incohérents détectés et les états corrigés (à partir des informations récupérées lors de l'analyse). Pour gérer l'incertitude, une zone grisée doit être affichée par dessus la vue, pour tout l'intervalle pouvant contenir des incohérences (c'est-à-dire à partir du moment où une suite d'événements a été perdue, jusqu'au moment où un état cohérent est retrouvé avec certitude). Ces éléments visuels doivent être intégrés dans l'interface graphique de l'outil d'analyse de traces.

La **solution proposée doit être développée et intégrée à l'analyseur de traces**. Pour cela, l'environnement de développement commun à tous les contributeurs de l'outil sera utilisé. Cette partie consistera globalement à écrire le code pour automatiser la solution, et à l'adapter aux spécificités de l'outil, si nécessaire. Cette tâche est réalisée au fur et à mesure que la solution est construite, puisque cela permet de la tester dans un contexte réel, en l'appliquant à des traces industrielles.

3.2 Environnement de travail

Pour ce travail, nous avons développé la solution sous la forme d'un plugin Eclipse, puisque Trace Compass, l'outil auquel elle doit être incorporée, est une application Eclipse constituée d'un ensemble de plugins. Afin d'intégrer et de tester facilement notre module, nous avons utilisé l'incubateur. C'est un projet relié à Trace Compass, qui permet d'ajouter facilement de nouveaux modules d'analyses ou fonctionnalités en cours de développement. Cela permet d'avoir un projet distinct pour notre solution, bien maintenable.

Afin de développer la solution, il faut préparer l'environnement de travail [1] :

1. installer et configurer Eclipse
2. importer le code source de Trace Compass et de l'incubateur et ajouter les plugins à l'espace de travail du projet
3. configurer le projet
4. lancer le script de configuration d'un nouveau projet de l'incubateur
5. importer le nouveau plugin dans le projet

1. voir `https://wiki.eclipse.org/Trace_Compass/Development_Environment_Setup` pour plus de détails

Ensuite, le module d'analyse peut être développé, indépendamment des autres modules.

La table 3.1 récapitule les différents outils utilisés pour ce travail.

Tableau 3.1 Caractéristiques de l'environnement

Système d'exploitation	Fedora 26
Noyau Linux	4.16.11-100
LTTng	2.10.2 - KeKriek
Trace Compass	Development version [2]
Eclipse	Eclipse IDE for Eclipse Committers version Oxygen.3a (4.7.3a)
Java	openjdk version 1.8.0_171

Nous avons aussi utilisé la version d'essai de l'outil de profilage YourKit pour profiler la solution et trouver des pistes pour améliorer la performance.

3.3 Travail réalisé

Le travail a été séparé en plusieurs étapes. Tout d'abord a eu lieu une phase de prototypage, d'expérimentation avec Trace Compass. Ensuite, nous avons commencé par développer la première partie de la solution, à savoir la vérification de la cohérence et de la certitude. Puis, nous avons développé la deuxième phase, celle d'inférence des événements. Ensuite, plusieurs tests de performance ont été réalisés et des modifications ont été apportées aux modules.

Le code est disponible à : `github.com/MMartin5/events-analysis`

3.3.1 Module Babeltrace

Afin de pouvoir tester facilement notre solution, nous avions besoin de pouvoir supprimer manuellement des événements d'une trace. Cela permet d'avoir un moyen de contrôler les événements déduits, en les comparant avec les événements sélectionnés pour la suppression. Cet outil a été implémenté comme un plugin pour Babeltrace.

Babeltrace est un lecteur de traces CTF, permettant de les visualiser dans un terminal. Il peut aussi convertir des traces, et fournit une bibliothèque développée en C (il existe aussi un *binding* Python) pour manipuler une trace. La version de Babeltrace 2.0.0-pre4 introduit les plugins afin de développer de nouvelles fonctionnalités pour l'outil. Nous avons donc décidé

2. `https://git.eclipse.org/c/tracecompass/org.eclipse.tracecompass.git`

de créer un nouveau plugin pour supprimer un ou plusieurs événements d'une trace au format CTF et ainsi créer une nouvelle trace.

Le plugin est disponible ici : `https://github.com/MMartin5/trace-editor`

3.3.2 Utilisation de l'outil

Nous allons présenter rapidement comment utiliser notre solution.

Il faut, de préférence, utiliser une trace contenant au moins un événement perdu. En effet, les analyses correspondantes ne sont activées que lorsque la présence d'événements perdus est détectée. Il est toutefois possible de forcer l'exécution des analyses, même en l'absence d'événements perdus. Cela peut permettre de découvrir des états incohérents inconnus. Toutefois, ceci ne devrait pas constituer le cas d'utilisation normal.

Dans le cadre de ce travail, nous avons utilisé notre module Babeltrace pour générer des traces avec événements perdus à partir de traces d'origine complète.

L'utilisateur doit également fournir un modèle de son système, sous la forme d'une (ou plusieurs) machine(s) à états . Pour cela, il doit écrire un patron XML (*XML pattern*) (correspondant à la FSM) en suivant la syntaxe définie par les analyses XML dans Trace Compass. Ensuite, il doit importer le fichier, de la même façon qu'il ajouterait une nouvelle analyse XML (Trace > Manage XML analyses > Import).

Dans le patron, l'utilisateur doit définir une action d'initialisation pour les scénarios. Elle doit initialiser l'attribut du scénario, qui doit être unique à chaque scénario. Donc, cet attribut va dépendre de ce qui est modélisé par la FSM (par exemple, pour une FSM modélisant un processus, on utilisera le tid des événements). Cette action doit être appelée dans toutes les transitions pouvant mener à la création d'un nouveau scénario (donc les transitions depuis l'état initial de la FSM). Il faut également que chaque transition, depuis un autre état que l'état initial, possède une action vérifiant que le champ de l'événement actuel associé aux attributs de scénarios correspond bien à l'attribut du scénario actuel. Ce sont des opérations qui doivent être spécifiées manuellement pour le moment, mais on devrait envisager de les intégrer directement dans la définition de tout scénario dans Trace Compass.

Pour nos évaluations, nous avons développé la machine à états modélisant un processus, et celle modélisant un CPU. Nous avons utilisé la définition du gestionnaire d'états qui décrit une sorte de machine à états sous forme de règles *événement* $\Rightarrow$ *état*, ainsi que nos connaissances du fonctionnement d'un processus et d'un CPU, pour construire ces deux FSMs.

Nous avons créé deux vues :

Coherence View permet de visualiser les états incohérents. La vue fonctionne de façon similaire à la *Control FLow View*[3]. Lors de son ouverture (Window > Show View > Other > Coherence View), l'analyse XML correspondant à la FSM va être exécutée (l'utilisateur doit donc avoir préalablement importé le fichier). Les états incohérents sont indiqués en gris sur la vue. Les événements incohérents associés sont identifiés par des marqueurs. Lorsqu'un événement perdu est rencontré, tous les états deviennent incertains. Cela est indiqué par des marqueurs qui grisent chaque entrée de la vue, jusqu'à ce qu'on ait à nouveau atteint un état certain (fin du marqueur).

GlobalInferenceView crée une vue de type *Control Flow*, mais en prenant en compte les événements déduits pour la trace entière. Cette vue est accessible depuis la CoherenceView ; elle s'ouvre lorsque l'on clique sur le bouton de sa barre d'outils. Elle peut être mise à jour en sélectionnant de nouvelles valeurs pour les champs des événements déduits. En effet, certains champs d'événements déduits peuvent avoir plusieurs valeurs possibles de probabilité égale. Dans ces cas, il est impossible de choisir une valeur plutôt qu'une autre. On laisse donc à l'utilisateur le soin de choisir la valeur qu'il considère la meilleure. Dans la barre d'outils, un bouton permet d'ouvrir un dialogue de sélection de ces valeurs. Une nouvelle sélection va mettre à jour la Global Inference View.

Dans l'article qui suit, nous allons présenter la solution, et comment elle répond aux différentes tâches que nous nous sommes fixées ci-dessus.

3. La *Control Flow View* est une des vues principales de Trace Compass. Elle présente les états successifs de chaque fil d'exécution sur une nouvelle ligne, et lie les différents fils entre eux lorsqu'il y a une opération d'ordonnancement sur un CPU, ce qui permet de suivre l'exécution du système.

CHAPITRE 4 ARTICLE 1 : MISSING EVENTS INFERENCE DURING TRACE ANALYSIS

Authors

Marie Martin
École Polytechnique de Montréal
marie.martin@polymtl.ca

Michel Dagenais
École Polytechnique de Montréal
michel.dagenais@polymtl.ca

Keywords: Tracing, Trace analysis, Missing data, Finite state machines

Submitted to: Journal of Computer and System Sciences, 29 July 2018

Abstract

While execution tracing techniques allow for low-overhead information collection at multiple levels, trace analysis is becoming increasingly powerful and is used to help diagnose complex problems, especially on distributed systems. However, a high flow of events may overwhelm the bandwidth available to store the trace, and thus imply being unable to write every event in the trace. When retrieving and analyzing these traces, current tools are unable to take into account missing events and to detect ensuing inconsistent states. This is problematic, because the analysis results may contain errors, but the user is not aware of it. Our work offers a way to add some support for lost events during trace analysis. The proposed framework is divided into two parts, one that checks the consistency and the certainty of each computed state transition, and the other that attemps to infer the missing events for each inconsistency detected in the first stage. In order to do so, we use Finite State Machines (FSM) to model the traced system and check each event against this model. We use the FSM to infer missing events by computing the shortest path between the inconsistent state and the last consistent state. We show that our solution can be used to provide visual clues to help users be aware of potential inconsistencies in the results of the trace analysis, and to suggest possible missing events. In addition, since we are now able to recover from missing state information, this work is an important step towards the parallel analysis of traces, where the global initial state is not available during the parallel analysis.

4.1 Introduction

Tracing is a powerful technique used to diagnose computer systems. It relies on collecting events from an operating system or application at a fine level of granularity. A tracer is a tool that can register every event occurring in the system on which it is running, and write these events into a file, called a trace. Parallel architectures are very common, and the number of cores in processors is ever increasing. With more and more data to handle, trace size is getting bigger, and the problem of lost events may arise. Tracers often use circular buffers to store incoming events before they are written to the trace. As buffer memory is limited, a high volume of events may lead to overwriting some events, or discarding the newer ones (see Figure 4.1 for a schema picturing this phenomenon). This is a common problem that can be observed in tracers such as LTTng (Desnoyers and Dagenais, 2006), ftrace (Rostedt, 2008) (Linux), and ETW tracing (Windows) (ewt). One could increase the buffer size, if the system has enough memory, but there are always situations where the volume of events generated (detailed trace) may overwhelm the available bandwidth for writing the trace (slow disk or communication link). Furthermore, it is often not possible to block the application when the tracing buffer is full, either because this would unduly affect the behavior of the traced system, or because it may cause a deadlock for example when tracing the operating system. Thus, in many cases, lost events are unavoidable. When this happens, LTTng can detect that events must be discarded and record this information into the trace packet metadata. When the trace is ready, it is imported into a trace visualizer such as Trace Compass [1], which will parse it in order to run various analyses. When the discard information is read from the metadata, it will create a *report of lost events* in the form of a synthetic event, that will appear as a real trace event to any analysis.

This may happen in other contexts, especially in streaming applications. Every time the incoming flow is too large to be consumed fast enough, we could lose events, packets, *etc.* In this context, the problem is different because we could allow some lost events (*e.g.*, it would decrease the quality of a video but a user could tolerate this). This is not the case for trace analysis, because it will create inconsistencies in the results. Network protocols are subject to such issues, just like digital signal processing applications or any other data streaming application with variable bandwidth. We can also assimilate lost events to missing data. This is particularly useful when we consider the parallel processing of a sequence of data. Each piece of the sequence needs an initial state, which is the state generated from the previous piece. We do not yet have access to this data if we simultaneously start to process every piece in parallel. It would be very useful to be able to infer the initial state.

1. https://tracecompass.org/

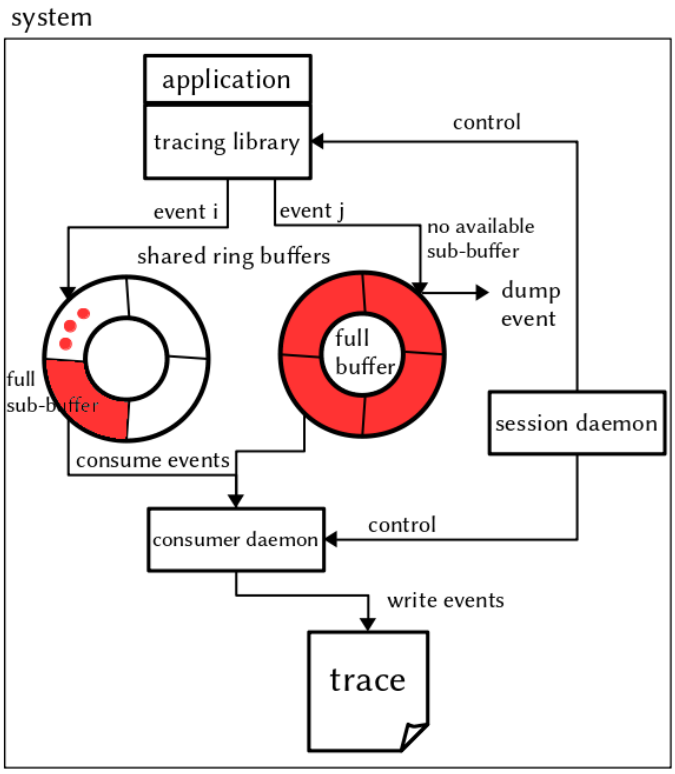


Figure 4.1 Schema of the simplified architecture for LTTng. Each recorded event is written to an available sub-buffer. When a sub-buffer is full, it is consumed by the consumer daemon that writes the events of the sub-buffer to the trace. If every sub-buffer is full and not consumed yet, the event has to be dumped (or the sub-buffer can be emptied – not depicted here).

Packet loss recovery is a well-studied issue (Perkins et al.; Bouloutas et al., 1991; Viterbi, 1967), but applications are mostly for binary data. It does not take into account the context of the error, though we have access to this information in a trace. In every event, there is useful data that should be exploited in order to infer missed states. Diagnosability and observability are also interesting concepts from the DES field (Sampath et al., 1996; Fabre, 2013) but unfortunately, the fact that lost events can happen anytime and with any event type prevents us from using these methods. While this topic offers a rich literature, syntactic and lexical analysis (Aho and Peterson, 1972; Boullier and Jourdan, 1987) is limited (fault models, recovery rules, etc.) and cannot take advantage of the information contained in a trace.

The solution that we propose will help to handle lost events. In particular, we focus on trace

analysis, where there is no support for that kind of problem to date. We are using state machines to detect inconsistencies in the trace, then to infer events in order to recover the ones that have been lost. We take advantage of the knowledge that the trace should follow a pattern matched by the FSM modelling the traced system. We also use several properties of state machines to deduce information from the trace events.

Our work is divided into two main parts. First, we focus on checking the consistency of the states that are generated from feeding the events to the state machine, as losing events means that the states may not have been correctly updated. The module responsible for trace analysis is based on a state machine that is taking events as inputs and generates state changes as output. However, the related FSM is not complete and insufficiently defined. For the purpose of prototyping our solution, we defined our own state machine, using a custom XML-based language. We can easily detect inconsistencies when trying to find a transition from the current state but finding one that could only be triggered from another state. This gives us a hint that we missed some state transitions from the current state to this new state. The second part is our strategy for inferring events, which is to compute a path in the FSM between these two states. That way, we can suggest a recovery from the inconsistency. After inferring the event (type), we need to infer its content. To do so, we use information from the state system, other events, and probabilities.

Our main contributions are:

— formal definition of the inconsistency and the certainty for a trace in the context of state machines used as models of the system (thus describing the set of expected sequences of events)

— proposing an algorithm to infer missing events (their type and content)

— proposing an algorithm to check the consistency of each event in relation to a state machine

— providing visual information to the user for accurate trace analysis

In Section II, we introduce the related work. Section III presents the motivation behind our work in further details. Our proposed solution is described in Section IV, and we apply it to some use cases in Section V. Finally, we evaluate our solution in Section VI and conclude in Section VII.

4.2 Related Work

The question of lost events has never been specifically addressed in the tracing literature. However, we can find a lot of relevant content in several related areas, such as lexical and syntactic analysis, Discrete Event Systems (DES), and network protocols, as state machines are used in all of these areas.

Matni and Dagenais (2009) showed that trace analysis can be performed using FSM. The work by Wininger et al. (2017) and Kouame et al. has laid the basis for our solution, by allowing the user to define a model of the traced systems in an XML-based language, using FSM. However, this prior work is only used to provide flexible user-defined analyses. It does not take lost events into account, nor is it checking the consistency of the states computed by the FSM.

A DES is a dynamic, time-invariant, nonlinear system (Cassandras and Lafortune, 2008). Its two main properties are that the transitions are event-driven and its state space is a finite set (hence called discrete). According to this definition, a computer information system is a DES (Sampath et al., 1996). Error diagnosis for DES is based around the notion of I-diagnosability (Sampath et al., 1996), stating that if every error can be found after observing a certain finite number of events starting from an indicator event, then the system is I-diagnosable. We can then construct two automata, the observer and the diagnoser, whose actions set can be divided into two subsets: the observable actions, and the unobservable actions (the latter corresponding to missing events) (Fabre, 2013). These automata are used to find if there could be an unobservable action (*i.e.*, an error) among a given sequence of actions (*i.e.*, a trace). These methods use a fault-free model of the system, which we already have, given our XML FSM. Unfortunately, our model is not I-diagnosable, because there is a cycle of unobservable events, as every event may happen to be lost. This can be formally proved using the twin-machine method, which we will not expand upon here.

Regarding network protocols, we identify missing events with lost packets. The mitigation of packet loss can be done in an active way, *i.e.*, packet retransmission, or in a passive way (Perkins et al.). In the latter case, corrective means are applied to the packets data, either with interleaving (data reorganization) or with forward error correction (additional information added to the packets). If these methods cannot be applied (*e.g.*, one-way channel and limited bandwidth), one could try to apply error correction from the receiver side, such as correcting codes. Bouloutas et al. (1991) propose an extension of the well-known Viterbi algorithm (Viterbi, 1967) for Error-Correction Parsing (ECP), supporting substitution and deletion, in addition to insertion. The authors built an estimator that computes the symmet-

rical of the operations on the FSM output so that it gives the best estimate of the original output.

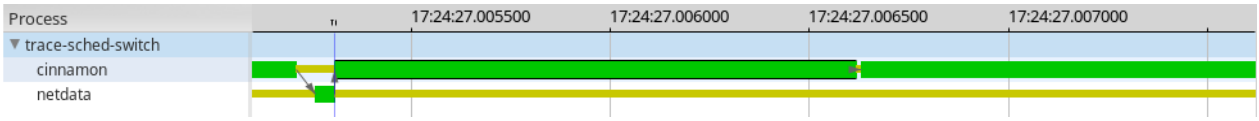
A missing event in a trace can be seen as a syntactic error, more precisely the deletion of a symbol. Several solutions have been proposed to deal with syntactic errors, in particular in the context of compilers. In (Aho and Peterson, 1972), the authors suggest extending the grammar by adding production rules for handling error cases, but it could become a tedious task if trying to cover every possible case. Moreover, it cannot deal with new, unknown errors. The method proposed by Boullier and Jourdan (1987) first tries to apply a local correction, then tries a global correction if unsuccessful. The authors are using corrective models for the local recovery, which is better than specifying every possible error, but still requires some effort from the programmer.

In business process management, the question of process conformance is studied. Given the model of a process (often represented as a Petri net), how can we check that there is no missing activity in a log? One solution is to read each event in the log and see if they fit with the model, which is called replay (Rozinat and van der Aalst, 2008). Adriansyah et al. (2011) use cost functions to assign costs to every deviation from the model, in order to find the instance of the model that best fits the log (*i.e.*, the instance whose cost is minimal). As the instances are constructed during replay, the A* algorithm is used to select the most fitting instances. Regarding recovery, Rogge-Solti et al. (2013) are using stochastic Petri nets as models, and compare them to logs following the alignment technique. Several possible paths are computed, and the most probable one is selected (using weighted transitions, probabilities on events, etc.). This method needs several readings of the log, which is impractical for large traces. Moreover, it cannot consider a path if no event within the log is on that path. Wang et al. (2013) aimed for a more efficient recovery method, thus proposing a backtracking algorithm. Instead of looking for every possible path, the authors use a branching technique to prune the branches that will not give an optimal solution.

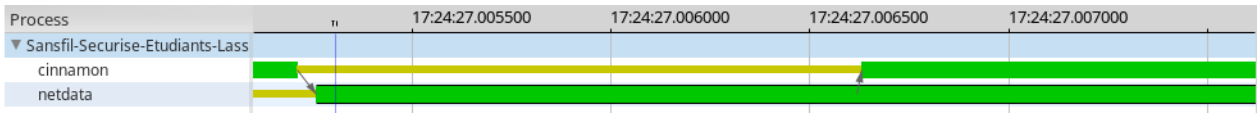
In general, the work mentioned above does not take into account the context of the errors. It cannot exploit the full potential of a trace, whose events contain useful information for checking the consistency of the states and reconstructing lost events.

4.3 Motivation

As mentioned in the introduction, tracing is a very powerful tool that can provide valuable information about the causes of performance problems, crashes, bugs, etc., to help with the diagnosis process. It is widely used and well integrated within the Linux kernel. It helps



(a) Trace without lost events



(b) Trace with a lost event (at T1)

Figure 4.2 A trace with two processes running on the same CPU, each process having one single thread. The green state means that the thread is running, the yellow state means that it is waiting for CPU availability. An arrow indicates that another thread is scheduled on the CPU.

to solve issues that could not be solved using other tools, such as a debugger or a profiler, especially in case of time-sensitive issues. Thus, improving the available tools is useful for a large variety of users and contributes to spreading such effective techniques.

It is essential to check the conformance of the model before beginning the analysis (Adriansyah et al., 2011). Likewise, we should check the conformance of the trace to the model of the system, otherwise we could end up with inconsistent results (though it would be right according to the analysis process for this given trace). Currently, to the best of our knowledge, there is no trace analysis tool that can support lost events. For example, Trace Compass does not take into account the fact that the states could have been updated, where data has been lost, and continues the analysis as is. By doing so, it gives no information to the user about the consistency of the analysis, except for indicating the area where events have been lost. This information is contained in traces using the CTF format (Desnoyers, 2010), where each event packet context has an optional field for the count of discarded events. With LTTng, each buffer keeps a counter of lost events that is updated when some events have to be discarded, and that is then written in the event packet context.

Yet, showing uncertainties and inconsistencies increases the understanding of the system execution. It allows users to be aware of possible errors during the analysis, and thus to select the most relevant information from the visualized results. Recovering lost events and getting the states back enables building a more robust tool.

To give an example of the kind of problems arising with lost events, we can take a look at Figure 4.2. On part 4.2a, we can see the original trace, without lost events. At `T1`, the process `netdata` is scheduled out, and `cinnamon` is scheduled in. If we delete the event at

`T1`, `netdata is never scheduled out`, which leads to a situation where two processes are running at the same time on the same CPU core when `cinnamon` is later scheduled in (see Figure 4.2b). We know that this cannot happen in a real system, but this is what is displayed in the view, because of the lack of support when facing such issues. A user could easily miss this inconsistency among the thousands of states, even though it is an important matter.

Some work has been done in order to parallelize the trace analysis (Reumont-Locke, 2015). The trace is sliced in several trace chunks using data partitioning techniques. While promising, the main problem remains that the analysis process requires access to the global state of the system. As the analysis is a sequential process, the computation that needs to be done at t is dependent on the global state computed at $t-1$. Here, each trace chunk is treated independently from the rest of the trace, so its analysis starts from an unknown global state. We need to be able to start the analysis without an initial global state, which will be possible if we are able to know when we reach an ascertained state.

4.4 Proposed solution

4.4.1 Basic concepts

Before presenting the framework, we will start with a few definitions.

An **event** is a piece of information obtained while tracing a system. It has a timestamp, a type (its name), and a content, which consists of one or several fields. A **field** is a pair of name and value. A **state** is described by a time interval and a value. Unlike an event that is punctual in time, a state has a duration in time. Several states can exist in the system at the same time, each describing a characteristic of the system or some related object. This characteristic is called an attribute. An **attribute** is the smallest subdivision of the model that can have a state. At every point in time, it has a unique state (if it has not been set yet, the value is null). It we get every attribute of the model at a time t, we get the global state of the system at that time.

From the previous characterizations, we define a **trace** as a sequence of events in chronological order, *i.e.*, $T = (e_0, e_1, e_2, .., e_n)$ where e_i is the ith event in the trace T of size n.

We assume that every event in the trace did happen. This makes sense, because an event is created when a tracepoint is hit, so it comes from a real action in the system. Moreover, the trace cannot be tampered with, so no event that did not actually occur in the system could have been added to the trace.

A trace can contain an artificial event created from extracting the discarded events count

field in the event packet context. As mentioned in the introduction, this field is generated by the tracer when it has to dump events, for instance when the buffer is full while new events are being produced, and triggers the creation of a synthetic event in Trace Compass. We will refer to it as `Lost events report`. This is used to indicate that at some point during tracing (this point being its timestamp), some events were lost. Ideally, the count of lost events is also part of the `Lost events report`, as is the case with LTTng, but this is not a strict prerequisite (some tracers may not be able to provide a counter).

In this work, we are using finite state machines (FSM) to model the system that is being traced. A **finite state machine** is mathematically defined as a quintuplet $(\Sigma, S, s_0, \delta, F)$ where:

Σ	the input alphabet, a non-empty finite set of symbols
S	the possible states, a non-empty finite set
s_0	the initial state, an element of S
δ	the state transitions function, such as $\delta : S \times \Sigma \rightarrow S$
F	the final states, a subset of S that can be empty

From this definition, we can easily see that an FSM can be converted to a graph. A state is a node, and a transition is an arc.

What we call the **label** of a transition is the event (*i.e.*, a word of Σ) triggering the transition. It should be mentioned that, in our case, the transitions can be conditional (*e.g.*, time constraints, specific attribute value...). In that case, the conditions are a part of the transition label.

A **scenario** is an instance of a given FSM. In our solution, there can be several instances of the same FSM. In other words, the FSM is the general model of an abstract object, and a scenario is the model applied to one specific instance of the abstract object. For example, if an FSM is used to model a process, then there will be one scenario per existing process in the system when it was being traced.

A scenario is identified by a unique identifier, related to an attribute of the object it represents. It is essential, because an event does not necessarily apply to every scenario. We can extract the information related to scenarios attributes from an event. For instance, the scenarios mentioned above would be identified by the tid of the process. Then, if the event is a `sched_switch`, it should apply only to the scenarios identified by `next_tid` and `prev_tid`

(fields from the event content).

Definition 1 (Consistency) Considering that we have read n events from a trace, so that the current state of a finite state machine F is S, then the $n+1$ event e is **consistent** with F, if there is a transition from S whose label is e, or there is no transition from any state.

e is consistent if $\delta(s_{current}, e) = s_p$ and $\delta(s_j, e) = \varnothing \ \forall j \neq current$; where $s_i \in S \ \forall i$

Given this definition, we can see that e is **inconsistent** if there is a transition in F whose label is e, from a state that is not S, and there is no possible transition from S with e as its label.

The definition is easy to understand considering the fact that if an event is in the trace, then it has been observed in the system during tracing. Considering a FSM, the event should either:

— be ineffective, *i.e.*, triggering no transition at all ; a given FSM uses only certain types of event, so it is acceptable for an event to have no effect on this FSM (for example, a FSM modelling a CPU will not be affected by events related to memory allocation operations, because they are not changing the state of the CPU)

— trigger a transition ; but the transition must be triggered from the current state, unless something went wrong, *i.e.*, (at least) one event was lost

Thus, if there is no transition from the current state s_1, but a transition could be triggered from another state s_2, it means that some intermediate transitions between s_1 and s_2 were not taken due to missing events in the trace. In that case, we can say that the event is inconsistent, as a way to flag this area of the trace.

Definition 2 (Certainty) An event e triggers a **certain** state S if e labels only transitions to S.

e triggers s certain if $\delta(s_{current}, e) = s$ and $\delta(s_i, f) = s_j$ with $f \neq e$ and $s_j \neq s$, $\forall i, j$ such as $s_i, s_j \in S$

This means that no transition to states other than S are labelled by e.

This definition is useful to determine if an event allows us to return to a global consistent state. After a period of lost events, every state becomes uncertain, because we do not know where these events occurred, so potentially every state could have been affected. We want to find the time at which we know that we are back to a consistent state. To do so, we

can use the fact that if a particular type of event always leads to the same state, *no matter from which state*, then we are sure to be back to a known consistent state, when this event is observed in the trace. Always leading to the same state means that the event labels only transitions to this particular state.

Definition 3 (Event inference) **Event inference** refers to the process of trying to retrieve lost events.

Each transition is labelled by an event type. Therefore, the objective is to find the sequence of states between the last known consistent state s_a (before the incoherence) and the first certain state s_b (after the incoherence). As we mentioned earlier, an FSM can be converted to a graph, so this becomes equivalent to looking for a path between a starting state (s_b) and a target state (s_a). The reader should note that we are backtracking in the FSM.

Definition 4 (Content inference) **Content inference** refers to the process of trying to retrieve the content of an inferred event. To do so, we use the following property:

If a transition occurs, then the conditions labelling this transition are verified.

When we are testing an event e with a transition t, we are trying to see if e can trigger the transition. This means that the event type of e should match the event type labelling t, and the condition(s) labelling t should be verified. Each condition can be about anything related to the state system and the event: current thread id, value of an event field, state value of a particular attribute, etc. When we infer a transition (and its associated event), we necessarily also suppose that every condition was met in order to be allowed to trigger that transition. Thus, when a condition involves the event content, we can make assumptions based on the value that this condition must have in order to be verified.

4.4.2 Framework

We propose a solution divided in two main parts, that we will introduce by presenting the general workflow. A trace, containing at least one `Lost events report`, and a model of the system, as a set of FSM, are the inputs for our solution. For the sake of simplicity, we will consider the case where we have only one FSM, but the solution is easily extendable to the case where several FSM are used. The first phase is the error detection phase, where we feed every event to the state machine and look for inconsistencies, if no transition is found from the current state. We also check the certainty of the triggered transitions. The second one is the error correction phase, which consists of inferring events for every inconsistency

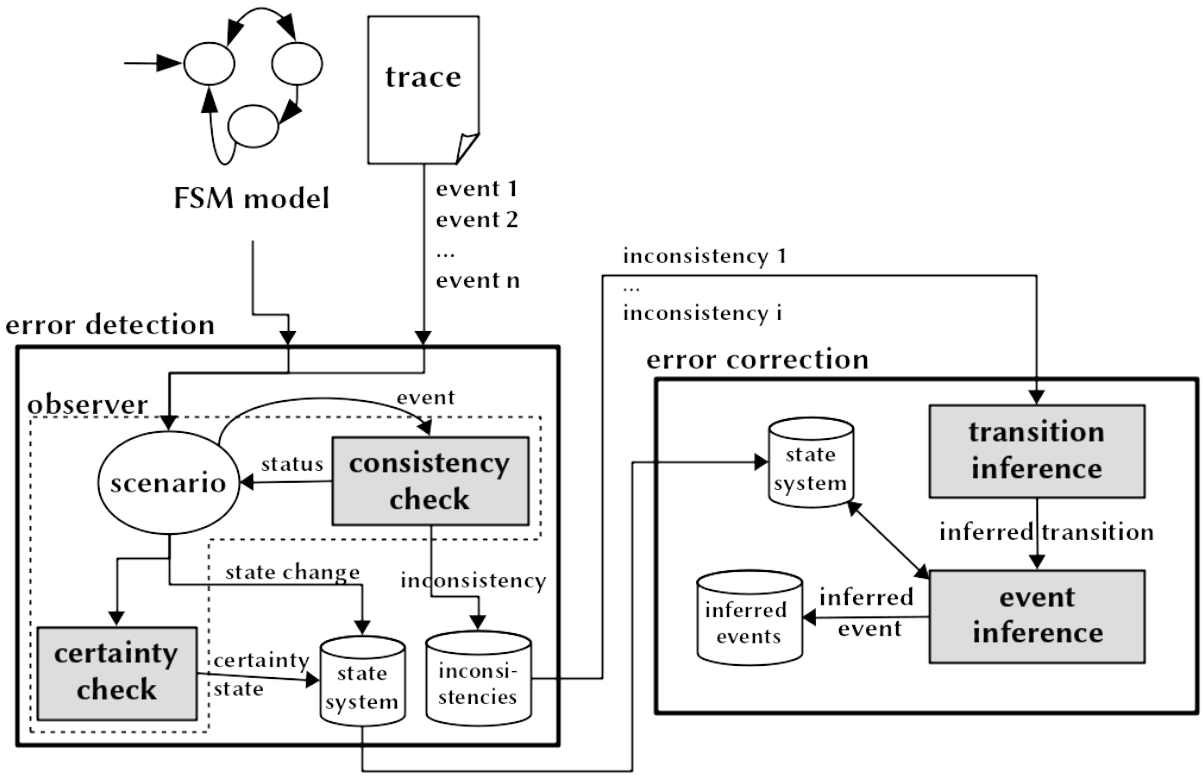


Figure 4.3 Proposed framework

found during the first step. To do so, we compute the shortest path between two states (the last consistent state and the target state), using statistics on transitions to weight the graph. Figure 4.3 sums up the proposed solution. We will now present these phases in details.

Error detection

Error detection consists of two distinct issues: consistency check and certainty check. We use the definitions given in subsection 4.4.1.

Using the FSM provided in the model, we need to build several useful data structures: `previousStates`, a map of the ancestors for every event (*i.e.*, states that we can leave when this event occurs), and `targetStates`, a map of target states for every event. Here, we define an event (the key of `targetStates`) as an event type combined with a list of conditions.

Consistency check The Algorithm 1 describes how to check the coherence of a given event, according to a given scenario. It is used in Algorithm 2, the more general procedure

that handles an event and checks its coherence (we will go into further details later). For each state `i`, we test every transition going out against the current event `e` and the conditions. If there is a match, and `i` is the current state of the scenario, then there is a transition from the current state, this is consistent and we can stop checking. But if there is a match, and `i` is not the current state, it means that a transition could have been taken from another state. This might be an inconsistency but we still have to check the rest of the states, so we only flag this as a possible problem and continue checking. Once every state has been tested, either we found a transition from the current state (and the algorithm returned true), or we found no transition from the current state and a transition from another state (so false will be returned), or no transition was found (in which case true is returned).

We only check among the possible previous states (using `previousStates`) for the current event, because no transition would ever be possible from a state for which the event cannot be a trigger. It allows for a lesser number of states to check, as opposed to checking for every state in the FSM. However, we must make sure that `previousStates` exists, because we might have no previous state if `e` is not accepted by `f`.

If the algorithm returns false, it means that an inconsistency has been found. In that case, we need to create an *incoherence*. An incoherence consists of:

- **the incoherent event** that prompted the creation of this incoherence
- **the attribute of the scenario** on which the event was found to be incoherent
- **the last known coherent event** before the incoherent one
- **the last known coherent state**
- **the inferred transitions** that is a list which will be filled later
- **the inferred events** that are associated with the inferred transitions

This algorithm runs in $O(|V|^2)$ in the worst case, where $|V|$ is the number of nodes (*i.e.*, states in the FSM). It corresponds to the case of a fully connected graph, where each node has a transition to every node (including itself); so there are $|V|$ transitions for each node. As we test every transition of every node, there is $|V|^2$ operations.

In order to know if a transition can be triggered, we need to compare its label to the event type, but also to look if its conditions are verified. The conditions associated with a transition can be arbitrarily complex, which explains why Algorithm 1 needs to test every transition. However, we can use a hash table to store the possible transitions associated with an event type. In that case, the hash table is built in $O(|V|^2)$, but this operation is done only once. Then, every lookup operation costs $O(1)$, and testing the conditions is $O(|V|)$, so the algorithm runtime becomes $O(|V|)$.

Algorithm 1: Check event

```
1  inputs
2      f : FSM
3      e : event
4      s : scenario
5  output
6      iscoherent : boolean
7  init
8      coherent ← true
9  start
10 prev ← previousStates[name of e] of f
11 if prev then
12     foreach state i of prev do
13         foreach transition t of i do
14             if (e matched for t) and (condition for t = true) then
15                 if i = current state of s then
16                     return true
17                 end
18                 else
19                     coherent ← false
20                 end
21             end
22         end
23     end
24 end
25 return coherent
```

We will now present the general coherence check mentioned earlier. The Algorithm 2 shows how to check the coherence of a given event, according to a given Finite State Machine (FSM). We need to test the event with the active scenarios related to this event and with the pending scenario (the scenario waiting to be started), or until we find one transition from any state which is not the current one. This is required because one event could trigger a transition in several scenarios. As soon as one transition could have been taken and is not, the event becomes incoherent (as described in Algorithm 1). The scenarios related to the event refer to the fact that an event does not trigger a transition in every scenario, so we can select only scenarios for which this could be the case. As mentioned in 4.4.1, the scenarios are identified by a unique attribute. With the method called `getAttributesForEvent`, we can extract the attributes related to scenario identifiers from an event (the implementation depends on the kind of identifiers used). This way, it limits the number of operations on

scenarios.

startChecking is set to true the first time we encounter a `Lost events report`, meaning we can start checking the consistency of the next trace events. There is no point in checking before that (unless explicitly asked to), because every event should be consistent.

We can stop checking as soon as we found as many transitions as required for the event (if this information is available). As explained in 4.4.1, events affect only a number of specific scenarios. For certain specific types of event, we know the maximum number of scenarios it could be applied to. Given this information, if we find as many transitions as the maximum number, we can stop the consistency check because the event will not affect other scenarios.

Certainty check We also need to check the certainty of the state in which the FSM is after handling the event. This is where the map `targetStates` will be useful. As we know from Definition 2, an event triggers a certain state if it labels only transitions to this state. Thus, in order to determine if `e` triggers a certain state, we have to check `targetStates` and see if there is only one state associated with the key `e`. If that is the case, we can mark this state as certain. In addition, there is a mechanism to mark every state as uncertain as soon as a `Lost events report` is observed.

The Algorithm 3 is a shortened version of the algorithm to handle an event. It shows only the steps where it computes information about the certainty of the state for each scenario, *i.e.*, after a transition has been triggered or after a `Lost events report`. Also, note that every state is initialized as uncertain, because we have no information about the states before starting to handle events.

This algorithm runs in O(n), where n is the number of scenarios, because it is a simple access to a map (search in a hash table is O(1)).

Error correction

After the error detection phase, we get a list of incoherences, associated with a list of possible valid transitions for each incoherence. We first need to compute the list of inferred transitions for each incoherence, then to create inferred events for each inferred transition.

Transition inference For each incoherence, we need to infer the supposedly lost transitions. First of all, we need to select the most probable transition t_0 from the list of possible transitions. To do so, we suppose that the most probable transition is the most frequent one. Using statistics collected during the analysis, we can easily access that information. Other

Algorithm 2: Coherence checking

```
1  inputs
2      f : FSM
3      e : event
4      hasLostEvents : boolean
5      startChecking : boolean
6  output
7      iscoherent : boolean
8  init
9      coherenceCheckingNeeded ← false
10     transitionsCounter ← 0
11     transitionsRequiredCounter ← getTransitionsRequiredCount(e)
12 start
13 if startChecking = true then
14 |   hasIncoherence ← false
15 |   coherenceCheckingNeeded ← true
16 end
17 if e is a 'lost event' then
18 |   startChecking ← true
19 end
20 eventAttributes ← getAttributesForEvent(e)
21 foreach attribute attr of eventAttributes do
22 |   scenario ← f.activeScenariosList.get(attr)
23 |   transition = f.next(e, scenario)
24 |   if transition then
25 |   |   transitionsCounter ← transitionsCounter + 1
26 |   |   coherent ← true
27 |   |   update(scenario)
28 |   end
29 |   else if coherenceCheckingNeeded = true then
30 |   |   result ← Checkevent(e)
31 |   |   if result = false then
32 |   |   |   hasIncoherence ← true
33 |   |   end
34 |   end
35 |   if transitionsCounter = transitionsRequiredCounter then
36 |   |   coherenceCheckingNeeded ← false
37 |   end
38 end
```

```
for pending scenario s' of f do
    // same operations than for an active scenario
    if s' became active then
        f.activeScenariosMap = addActiveScenario(s', e, f.activeScenariosMap)
    end
end
if (startChecking = true) and (transitionsCounter != transitionsRequiredCounter) and
  (hasIncoherence = true) then
    return false
end
else
    return true
end
```

Algorithm 3: Check certainty

```
inputs
    f : FSM
    e : event
    ss : state system
    certaintyMap : map(pattern, set(string))
init
foreach active scenario s of f do
    ss.certaintyStatus[s] ← uncertain
end
start
foreach active scenario s of f do
    if e is a 'Lost event' then
        ss.certaintyStatus[s] ← uncertain
    end
    transition ← f.next(e)
    if transition then
        if size(certaintyMap.get(e)) = 1 then
            ss.certaintyStatus[s] ← certain
        end
    end
end
```

metrics or criteria could be used to estimate the most probable transition, but we choose a simple one for this solution.

Then, we need to find the other transitions necessary to reach the last known coherent state. The origin of t_0 becomes our starting point, and our target is the last known coherent state.

The Algorithm 4 computes a list of transitions between two states. To achieve this objective, the shortest path from one state (starting node) to the other (targeted node) is computed, because it can be assimilated to the most probable transitions that occurred and that were lost. We use Dijkstra's algorithm (Dijkstra, 1959) for computing the shortest path. The FSM can be easily converted to a graph for the purpose of this algorithm. The map `transitionCounters` is constructed by incrementing the counter (value) associated with the transition (key), each time this transition is taken (we observe this while reading the trace). The values used to weight the graph are from this map.

First, we initialize the current node `current` with start. We need a map `distances` to save the tentative distance from each node to the start (which is initialized to infinity because the distance is unknown, except for the start node, whose distance to itself is 0), a map `prev` to register the ancestor node on the shortest path for each node (initialized to `UNDEFINED`), and a list of unvisited nodes to insure that we check every node only once.

Considering the current node, we check every transition leading to a neighbor node. We evaluate the path to target using statistics on transitions. If the new computed distance is smaller than the one registered in `distances`, then we found a new shorter path from this node. After every neighbor is processed, we look for the next node to set as the current node, which is the one whose tentative distance is the smallest. We go on until the target is visited. We can finally compute the shortest path by following the nodes in the `prev` map, from target to the start (*i.e.*, the first `UNDEFINED` value that we reach).

The algorithm runs in $O(|V|^2)$, where $|V|$ is the number of nodes. This is the time complexity for the original version of Dijkstra's algorithm. There is an alternative version of this algorithm that runs in $O(|E| + |V|log|V|)$ where $|E|$ is the number of edges, using a priority queue (implemented with a Fibonacci heap). Here, we chose the original, simpler version, since V is typically small.

We will not reproduce here the proof of correctness for Dijkstra's algorithm, but the interested reader can find it in (Cormen et al., 2009).

Event inference The second part is to infer events. For each inferred transition, we look at the label to know the event type. Then, we need to infer what the content of this event is. To do so, we see if the label also contains some conditions. If there are, we select the conditions that are about event fields, *i.e.*, conditions specifying that a given event field should take the defined value. The fact that this condition must have been verified for the transition to be taken (see Definition 4) allows us to compute a value for the event field, based on the value that this condition indicates it should take.

Algorithm 4: Compute missing transitions

```
1  inputs
2      start : string                                          // the starting node
3      target : string                                         // the targeted node
4      F : fsm
5      stateNeighbors : Map < string, Set < transition >>
6      transitionCounters : Map < transition, long >
7  output
8      inferredTransitions : List < transition >
9  init
10     current ← start
11     unvisited ← set < string >
12     distances ← map < string, float >
13     prev ← map < string, transition >
14     foreach state s of F.states do
15     |   distances[s] ← INFINITY
16     |   unvisited ← s
17     |   prev[s] ← UNDEFINED
18     end
19     distances[current] ← 0
20 start
21 while target is in unvisited do
22     currentDist ← distances[current]
23     neighbors ← stateNeighbors[current]
24     foreach transition t of neighbors do
25     |   neighbor ← t.from
26     |   if t is in transitionCounters then
27     |   |   weight ← transitionCounters[t]
28     |   end
29     |   else
30     |   |   weight ← 1
31     |   end
32     |   newDist ← currentDist + 1/weight
33     |   if distances[neighbor] > newDist then
34     |   |   distances[neighbor] ← newDist prev[neighbor] ← t
35     |   end
36     end
37     remove current from unvisited
38     foreach state remainingNode in unvisited do
39     |   current ← min(dist[remainingNode], dist[current])
40     end
41 end
```

```
38 transitions ← Stack < transition >
39 node ← target
40 while prev[node] != UNDEFINED do
41 |   t ← prev[node]
42 |   transitions.push(t)
43 |   node ← t.to
44 end
45 return transitions
```

For example, we can take a look at this XML condition labelling a transition t_1:

```
<test id="prev_state_0">
  <if>
    <condition>
      <field name="prev_state" />
      <stateValue type="long" value="0" />
    </condition>
  </if>
</test>
```

Given this declaration, every time t_1 is inferred, we suppose that this condition is true. This means that there should be a field called `prev_state` whose value is `0` in the inferred event, so its content can be updated with that information.

This is a pretty straightforward example; most cases are more complex but the idea is the same. Sometimes, multiple values could fit the desired state value. We can compute probabilities based on some information from the state system to select the most likely one. We can also let the user decide which value to select (he may have some intuition about what has been lost, and the user knows the system better than a generic trace analyzer).

4.4.3 Implementation

Our solution, that we call *event-investigator*, has been implemented in Java for Trace Compass. It has been integrated into this tool as a module, but should be merged later into the main components. The work is available on GitHub at: `https://github.com/MMartin5/events-investigator`.

The user should have a trace with lost events and an XML file containing the FSM that will be used for the consistency check, as defined using the representation format provided by

Trace Compass for XML analysis. For now, the user has to manually define the XML model, but it is not checked, except for syntactic errors (validation of the XML file). Modeling errors in the user specified FSM may therefore remain undetected. We could automate this step by computing the model from a set of traces for the system (for example, see the work (Koskimies and Mäkinen, 1994)). Since it was not an issue for the prototype, we did not implement this solution. We are using LTTng traces in CTF format, but any trace format providing events (from which the states will be computed) can work.

Once the XML file is fed to Trace Compass, the XML model is parsed to generate the FSM (a Java object in Trace Compass). During this process, we need to build the necessary data structures, mentioned in 4.4.2, such as `previousStates` and `targetStates`. There is an attribute in the trace that indicates the presence or absence of lost events. If there are some lost events in that trace, we activate the creation of scenario observers. We will explain this in the following paragraph. The creation of observers can be forced by the user if he so wishes, but we consider here the general case where he genuinely wants to check a trace.

After parsing the XML file, the analysis can start. The trace is read one event after the other. Each event is sent to the *state provider*, that sends it to the *event handler*. The event handler has to manage the FSM and start the scenarios. It sends the event `e` to every active FSM, that will then redirect it to the appropriate scenarios. To do so, the FSM will compute a list of scenario attributes for `e`, by extracting relevant values from its content. These relevant values are user-defined in a class file, that indicates which event fields should be used for a given type of model. It is a flexible way to support attribute extraction for any model. For every attribute, we select the scenario whose id value is the attribute.

The scenario observer is a special class of scenarios. It works as a classic scenario but is enhanced in order to be able to check the consistency of the events. It implements Algorithm 1.

For each event `e`, if an inconsistency is found, we register `e` as an incoherent event. By doing so, it creates an *incoherence* object and saves the list of possible repair transitions that were computed during the check.

The second phase (error correction) is performed on demand, after the analysis, because a user does not necessarily want to compute corrections for the inconsistencies; he may only wish to check if there are some. Incoherences are collected from the previous analysis. Once the events are inferred, we create a special trace called *Inference trace* to save them. This is an artificial trace constructed on top of the original trace (wrapper), to which we add the inferred events, only used later by a view.

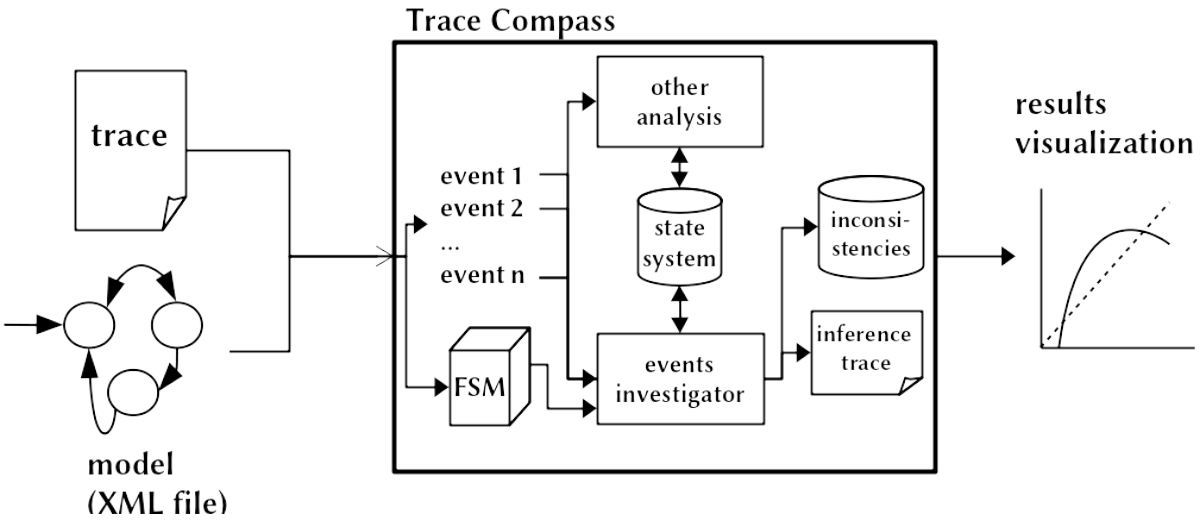


Figure 4.4 Framework viewed as a black-box inside Trace Compass

Figure 4.4 gives a broad summary of this solution.

4.5 Use Cases

We will present three use cases, in order to demonstrate the usefulness of different features (uncertainty markers, inferred events, and inconsistencies).

4.5.1 Snapshot

In LTTng, there is a mode called *snapshot mode* that can be selected during the configuration of a tracing session. Basically, a snapshot is a copy of the current content of the buffers when the snapshot is requested. Usually, a user wants to continuously trace a system. A classic tracing session is created and a state dump happens, which allows for building an initial state. However, the user may want to reduce the amount of recorded data, or to look only at the latest events before a specific time. With the command `lttng snapshot`, the user can ask for a snapshot of the current tracing session buffers to be written to the trace file. When the tracing session is created in snapshot mode, it does not guarantee that the state dump will still be contained in the buffers when the snapshot is actually requested. If the state dump events have already been overwritten, then the snapshot trace does not have an initial state.

As a consequence, snapshots are a specific case of lost events. We can consider that every

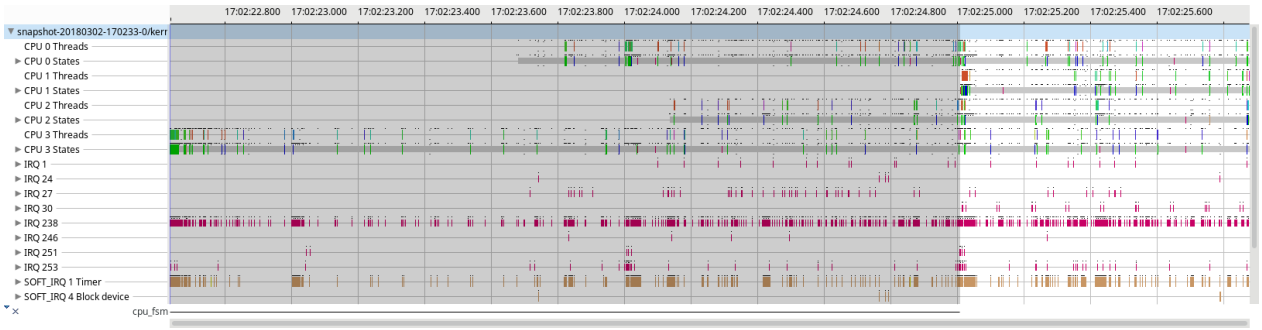


Figure 4.5 Uncertainty markers (grey area) on the resources view, which displays the state of each CPU and each IRQ data structure on a new line. The grey area represents the interval where the states are uncertain. Here, we only see one because it is associated with the last CPU for which we obtain information about the certainty of its state.

previous event has been lost. This is problematic because inconsistencies may appear during the trace analysis. As we are not registering events from the beginning of the tracing session, the current content of the buffers may correspond to different time periods. One buffer could receive a very high flow of events, so that every event in this buffer is fairly recent, while another one could receive very few events and never having been emptied, so that events date back to the start time of the trace. Therefore, when building the state system, we cannot be sure that the first states computed with events from the high-flow buffer are consistent, as we "lost" some past events. We cannot even be sure that some states start late because of this phenomenon, or simply because there was no activity for the objects related to these state models.

To guarantee that the snapshot will reflect the actual state of the buffers, the user should stop the tracer before recording a snapshot. Otherwise, when the snapshot is requested, the content of every buffer will be saved in the trace, which could take some time, depending on the size of the buffers. In the meantime, the system is still running and, if the tracer is running too, some events in the buffers could be overwritten by new recorded events, while the snapshot is being written. This is why the recommended practice is to stop tracing before taking a snapshot, since existing analysis tools have no support for lost events and consistency checks. With our solution, we could imagine dealing with such a case. This was not exercised in this work, but it would be easy to implement.

We will see in this example how we can use the certainty check to show where we are certain to have a global consistent state. This will tell the user at what point he can trust the results of the analysis.

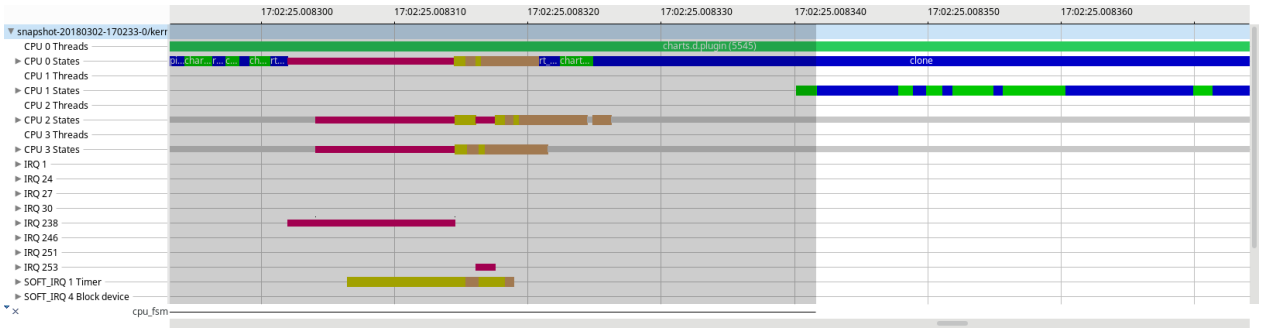


Figure 4.6 Uncertainty markers on the resources view – zoom. Each color is for a specific state (green for running, blue for system call, pink for interruption, orange for soft irq, yellow for raised soft irq).

Trace `snap-session-cafe-20180302-160841` has been created using the snapshot mode, with all kernel events enabled (the list of commands used to generate the trace is available on GithHub).

Once the trace has been generated, we imported it into Trace Compass and started the consistency analysis. Here, we are not looking for inconsistencies, as there is no lost event, but instead we want to get information about the certainty of the computed states. Figure 4.5 shows the resources view, which displays the CPUs states and the IRQs. The grey marker represents the area where there is some uncertainty.

We can see from this figure that the interval from the start of the trace to the time around 17:02:25 is uncertain. This can be explained when zooming further, as displayed on the Figure 4.6. We can see that the global state becomes certain when CPU 1 starts. Because CPU 1 is the last CPU for which we receive some information, we cannot determine if the state is certain before the first event occurring on that CPU. That is why the whole area is grayed, indicating to the user that he should be careful when considering that part of the analysis. When navigating the view, the user does not have to check every state to know if the observed area contains certain states, he can see it right away.

This does not depend on the results of the consistency check. What we call *consistency analysis* is the certainty check and the consistency check (as they were implemented at the same time), but actually these are two distinct concepts that do not influence each other.

Note that the certainty is not updated by the first state of CPU 1, which seems incorrect, but actually this is because of the definition of the certainty (4.4.1). It means that the first state has been triggered by an event that does not lead to a certain state.

4.5.2 Manually deleted events

One of the many advantages of traces is that we can easily manipulate them for testing purposes. In particular, we can manually delete events from a trace. That is very useful in order to test that lost events are correctly retrieved. Indeed, we have full control on which events have been deleted for testing purposes, therefore we know what the inferred events should be.

In order to do so, we developed a *babeltrace* plugin that is able to write a new trace from an existing LTTng trace and to remove some events while doing so. Our plugin is called `trace-editor`. The user provides a path to a trace and a list of events (*i.e.*, position numbers) or an interval of events that should be deleted. The trace generated by the plugin is the original trace from which the specified events have been deleted. Information is added to the packet context in order to be able to create a `Lost events report` in Trace Compass.

The trace `trace-delete-100-109` has been generated using the command:

```
babeltrace Traces/trace-sched-switch/
--component sink.trace-editor.editor
--name editor --path trace-delete-100-109
--params 'delete-interval="100:109"'
```

where `trace-sched-switch` is the existing trace. This trace was created with only `sched_switch` events enabled. The events that have been deleted (from event 100 to event 109, inclusively) are listed in Table 4.1.

Figure 4.7 shows the consistency view for the trace with deleted events. The first thing that should be noticed is that every state becomes uncertain when the `Lost events report` starts. As soon as a state becomes certain, the marker stops (that happens at a different time for every state). Moreover, the user can see a red marker, indicating that an inconsistent event was found. The state that led to this event is then grayed, because an inconsistent

Table 4.1 List of the events deleted from the original trace

	Timestamp	CPU	Event type	Contents	TID	Prio
100	17:24:26.540 643 291	1	sched_switch	prev_comm=gnome-terminal-, prev_tid=5078, prev_prio=20, prev_state=1, next_comm=swapper/1, next_tid=0, next_prio=20	5078	20
101	17:24:26.541 686 229	1	sched_switch	prev_comm=swapper/1, prev_tid=0, prev_prio=20, prev_state=0, next_comm=gnome-terminal-, next_tid=5078, next_prio=20	0	20
102	17:24:26.541 720 405	1	sched_switch	prev_comm=gnome-terminal-, prev_tid=5078, prev_prio=20, prev_state=1, next_comm=swapper/1, next_tid=0, next_prio=20	5078	20
103	17:24:26.542 398 405	0	sched_switch	prev_comm=swapper/0, prev_tid=0, prev_prio=20, prev_state=0, next_comm=kworker/0:4, next_tid=20850, next_prio=20	0	20
104	17:24:26.542 407 130	0	sched_switch	prev_comm=kworker/0:4, prev_tid=20850, prev_prio=20, prev_state=1, next_comm=ksoftirqd/0, next_tid=7, next_prio=20	20850	20
105	17:24:26.542 416 392	0	sched_switch	prev_comm=ksoftirqd/0, prev_tid=7, prev_prio=20, prev_state=1, next_comm=swapper/0, next_tid=0, next_prio=20	7	20
106	17:24:26.542 974 334	0	sched_switch	prev_comm=swapper/0, prev_tid=0, prev_prio=20, prev_state=0, next_comm=netdata, next_tid=1741, next_prio=39	0	20
107	17:24:26.542 982 955	0	sched_switch	prev_comm=netdata, prev_tid=1741, prev_prio=39, prev_state=1, next_comm=swapper/0, next_tid=0, next_prio=20	1741	39
108	17:24:26.544 393 264	3	sched_switch	prev_comm=swapper/3, prev_tid=0, prev_prio=20, prev_state=0, next_comm=rcu_sched, next_tid=8, next_prio=20	0	20
109	17:24:26.544 398 692	3	sched_switch	prev_comm=rcu_sched, prev_tid=8, prev_prio=20, prev_state=1, next_comm=swapper/3, next_tid=0, next_prio=20	8	20

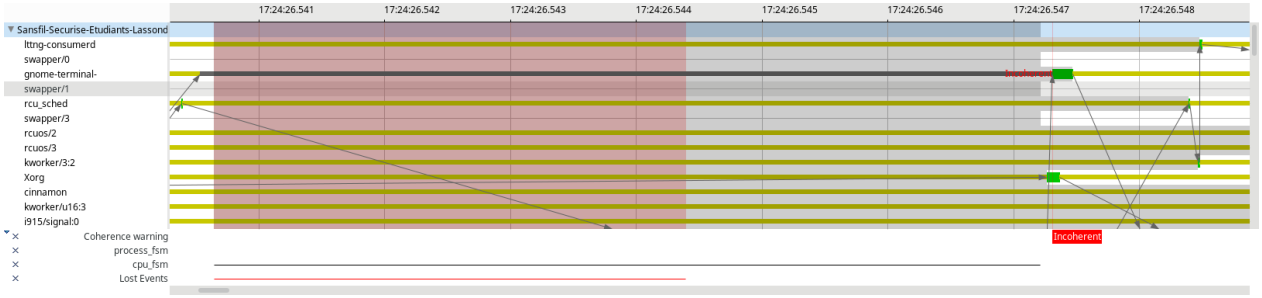


Figure 4.7 Consistency view for `trace-delete-100-109`. This view shows the state of each thread on a new line, with the information about its consistency and certainty. The red marker indicates the interval where events were lost, whereas grey markers indicates when states are uncertain (one marker per thread). The latter all start when the lost events interval starts, but each stops at a different time, depending on when a certain state is reached. The dark state represents an incoherent state, to which a red *Incoherent* label is added at the bottom of the view.

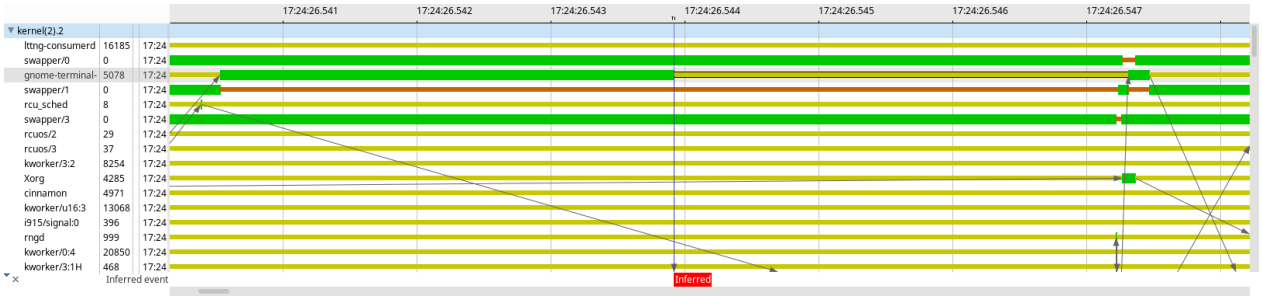


Figure 4.8 Inference view for `trace-delete-100-109`. This view displays the state of each thread on a new line, but these states have been computed with the addition of the inferred events to the original trace. This explains why the states shown here are slightly different from the previous view. The red label at the bottom of the view indicates an inferred event (at the time where we infer events were lost).

event is the evidence of some missing state changes before. This is the information available to the user to estimate the validity of the trace.

The inconsistency comes from the fact that the event 100 was deleted, thereby the process `gnome-terminal` is not scheduled out of CPU 1. When this process is later scheduled in (event 116), it creates an inconsistency because it is already running. A process already running should not be scheduled in, as defined in the model.

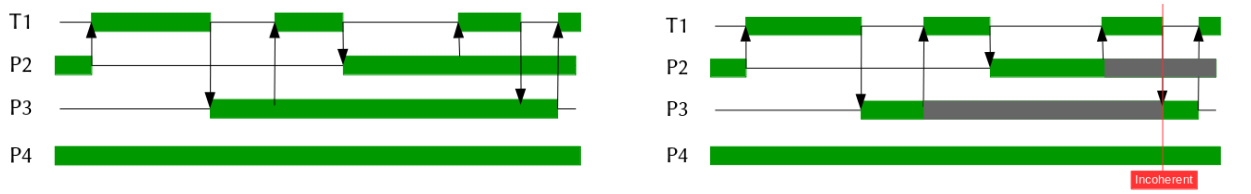
If he chooses to do so, the user can open the inference view, which is shown in Figure 4.8. It computes and displays the inferred events. On this view, we can see a red marker "*Inferred*" that indicates the location of the inferred event related to the inconsistency considered previously. This event is a `sched_switch` event, which schedules the process out. We can check in the original trace that this is indeed the missing event.

Unfortunately, our solution is not able to infer the rest of the deleted events, because they do not create any inconsistency. This is a limitation of our work. If, during the lost interval, the events create a loop of states (*i.e.*, there are some state changes but we get back to the state before the interval of lost events), then it cannot be detected by the consistency check. Nonetheless, there will be an uncertain marker to show that there *may* be an inconsistency, because after the lost events interval, no state change occurred that could lead to a certain state. That way, the user is still informed of potential issues.

4.5.3 Industrial trace

Here we will see how this work has been applied to an industrial trace. It corresponds to a real problem encountered in an industrial work environment. The work context is that of embedded systems. Due to limited resources and constraints on the execution time, the user wants to minimize the required tracing, thus minimizing the overhead. In that context, only one thread is being traced in the system for scheduling-in/out operations (`sched_switch` event type), *i.e.*, we only write `sched_switch` events coming from or to this specific thread. This obviously leads to inconsistencies in the results of the trace analysis, because we did not record the scheduling information for the other threads. In that specific case, the events were not lost, but simply not generated (tracing was not activated for all threads). Due to confidentiality concerns, we do not reproduce the trace here. We use a synthetic representation of this trace, which shows the main interesting points. The thread that is being traced is identified as `T1`.

In this system, there is only one CPU, so we expect to see only one green state (*i.e.*, running state) at every point in time. This is not what happens, as can be observed in Figure 4.9a,



(a) Control flow view for a representative industrial trace. Similar to the consistency view and the inference view, it displays the state of each thread on a new line (green for running in usermode, nothing for waiting for CPU)

(b) Consistency view applied to a representative industrial trace. The grey state is for pointing out the inconsistent state.

Figure 4.9 Industrial trace use case

because of the missing scheduling events. This is a very confusing view for a user that is trying to learn about the system, it looks like there are several CPUs and, if the user is not aware of the peculiarity of this trace, he could draw incorrect conclusions.

This is a case where our consistency view comes in handy. Figure 4.9b shows what the results of the consistency analysis look like, with the uncertainty markers disabled (because we have no lost events, this is not a very interesting feature here). The point here is to make the user aware of what he sees in the view. It is not necessarily useful to try to infer events either.

Of course, in the real view, there are many inconsistencies, so much that the readability is poor at the end of the trace (we will go into further details later). What should be noted is the fact that we now have only one visible running state at any given time, thanks to the incoherent grey state (except for thread `P4`, that could not be detected as inconsistent because there is only one event that triggers its running state, and that is the first event of the trace).

Our solution addresses real industrial issues. It provides relevant information, used to get a better understanding of the trace analysis results.

4.6 Evaluation

4.6.1 Methodology

All the tests have been executed on a Dell Inspiron 3656 64 bits quad-core machine, with an AMD A10-8700P Radeon R6 CPU, 12G RAM, running Fedora 26 with a 4.16.11-100 Linux kernel. The Eclipse version used is 4.7.3a (Oxygen.3a Release), with Eclipse IDE for Eclipse

Committers.

We have been running our tests with JUnit4 and the PerformanceMeter class from the test module of Eclipse [2]. Each test is run for 25 iterations, for each trace.

There are some optimizations performed by the JVM that we do not want to take into account in our results, such as the JVM warmup (this refers to the time it takes for the JVM to find hotspots that need optimizing, at the beginning of the benchmark). Out of 25 iterations of our benchmark, we selected only the last 15, in order to avoid variations from the warmup. We also ran each benchmark in a separate process, so that no optimization done on a previous benchmark affects the following operations. It should be noted that the results of JUnit benchmarks vary greatly from one run to another, especially for small traces, as operations out of our control (such as garbage collection) can happen and increase the runtime.

We studied the overhead introduced by our solution. The consistency check has been applied to every event (instead of starting when reading a `Lost events report`) in order to evaluate the cost of the solution for the whole trace. We also compared two algorithms for the consistency check. The *check optim.* corresponds to the algorithm 1, and the *check naive* corresponds to a similar algorithm that does not try to select the previous states (ancestors) prior to the consistency check, but instead tests the event on every existing state.

The overhead of the consistency check is calculated as

$$\frac{time_{withcheck} - time_{withoutcheck}}{time_{withoutcheck}} * 100$$

. To know the cost of our solution, we compared the execution time of the analysis with the execution time without, divided by the execution time without to get a ratio. In a similar way, the overhead of the event inference is

$$\frac{time_{withcheck+inference} - time_{withcheckonly}}{time_{withcheckonly}} * 100$$

.

Two FSMs have been used for these analyses, one for modelling a CPU, and the other one to model a process. The corresponding XML file is available at: `https://github.com/MMartin5/events-investigator/blob/master/docs/common/kernel_analysis_from_fsm.xml`

2. http://archive.eclipse.org/tracecompass/doc/stable/org.eclipse.tracecompass.doc.dev/Performance-Tests.html#Performance_Tests

Table 4.2 Execution time of the use cases with or without event inference (with [standard deviation])

	use case A	use case 2	use case 3
no check (s)	3.69 [3.27, 4.1]	1.27 [1.24, 1.3]	1.24 [1.21, 1.26]
check naive (s)	6.64 [6.42, 6.86]	1.38 [1.35, 1.41]	1.25 [1.22, 1.28]
check naive + infer (s)	6.97 [6.85, 7.09]	1.39 [1.35, 1.43]	1.29 [1.27, 1.32]
check optim. (s)	3.75 [3.61, 3.9]	1.39 [1.36, 1.43]	1.26 [1.23, 1.28]
check optim. + infer (s)	3.76 [3.66, 3.87]	1.39 [1.36, 1.43]	1.27 [1.24, 1.3]
overhead consistency check with naive (optim.) (%)	79.95 (1.63)	8.66 (9.45)	0.81 (1.61)
overhead inference with naive (optim.) (%)	4.97 (0.27)	0.72 (0)	3.2 (0.79)

4.6.2 Use cases evaluation

We first look at the execution time for each use case from 4.5. The results are presented in Table 4.2.

We can see that the optimized algorithm is better than its naive version. This is especially the case for the first use case, where the naive algorithm overhead is 79.95 %, whereas the optimized algorithm overhead is 1.63%. Thus, selecting appropriate states prior to the consistency check is important.

Depending on the use case, the overhead of our solution (considering the optimized version) can vary between 1.61% and 9.45%. When the event inference step is activated, it only adds less than 1% overhead. Overall, the additional cost is reasonable, considering the new information provided by our method.

4.6.3 Benchmark

Several factors can have an impact on the performance of our solution:

— the size of the FSM; the more states in the FSM, the more computations are required for each event that needs checking

— the number of lost events; the more lost events in a trace, the more inconsistencies created, the more events to infer

— the number of modelled objects in the system during tracing; *e.g.*, if our analysis is based on an FSM modelling a process, there will be a significant difference between a trace from a system with one thread and a trace from a system with a hundred threads, for a similar number of events

— the size of the trace; a bigger trace implies more events to handle

Table 4.3 Analysis execution time with or without event inference (with [standard deviation])

	trace 1	trace 2	trace 3	trace 4
size	164K	2.6M	14M	86M
nb. events	2188	40902	595641	2689393
nb. scenarios	134	456	169	365
nb. inconsistencies	1	3	308	714
nb. inferred events	1	3	769	1369
no check (s)	1.29 [1.27, 1.32]	3.05 [2.95, 3.16]	20.42 [19.68, 21.16]	49.34 [48.15, 50.53]
check naive (s)	1.37 [1.32, 1.41]	3.9 [3.79, 4.02]	24.99 [24.49, 25.48]	118.2 [117.6, 118.8]
check naive + infer (s)	1.39 [1.34, 1.43]	3.98 [3.86, 4.09]	25.8 [25.28, 26.33]	118.8 [117.6, 119.4]
check optim. (s)	1.34 [1.3, 1.38]	3.71 [3.63, 3.79]	22.14 [21.55, 22.73]	57.78 [56.82, 58.75]
check optim. + infer (s)	1.34 [1.31, 1.37]	3.85 [3.75, 3.96]	22.23 [21.54, 22.92]	57.94 [56.88, 59]
overhead consistency check with naive (optim.) (%)	6.2 (3.88)	27.87 (21.64)	22.38 (8.42)	139.56 (17.11)
overhead inference with naive (optim.) (%)	1.46 (0)	2.05 (3.77)	3.24 (0.41)	0.51 (0.28)

When the number of lost events increases drastically, the analysis may take a very long time. This may not be a real limitation in practice because, in such a case, it is recommended to generate a new trace, using larger tracing buffers or with fewer tracepoints activated. Indeed, a trace with too many lost events is more or less unusable.

We executed the benchmark on 4 different traces, using the same methodology than for evaluating the use cases. The results can be found in Table 4.3. We see that the bigger the trace, the longer it takes to run the analysis. The number of scenarios also has an impact on the performance. This can be seen by comparing the overhead of the naive method for trace 2 (27.87% for 456 scenarios) and trace 3 (22.38% for 169 scenarios). The most important factor remains the number of events in the trace (*e.g.*, trace 2 has more scenarios than trace 3, but the analysis runs faster).

As expected, the more events in a trace, the longer it takes to run the analysis. The overhead of checking the consistency can become quite important for very large traces, but we should keep in mind that we forced the analysis to be applied to the whole trace, which should not happen in the general case.

The event inference does not bring a lot of overhead (at most, 3.77%). This shows that we can get interesting additional information at low cost.

4.6.4 Accuracy evaluation

In order to evaluate the accuracy of our solution, we designed a simple dummy FSM (see Figure 4.10) and manually created an XML trace [3] with 63 events and 3 different instances

3. see `https://github.com/MMartin5/events-investigator/blob/master/usecases/eval/testTrace_eval.xml`

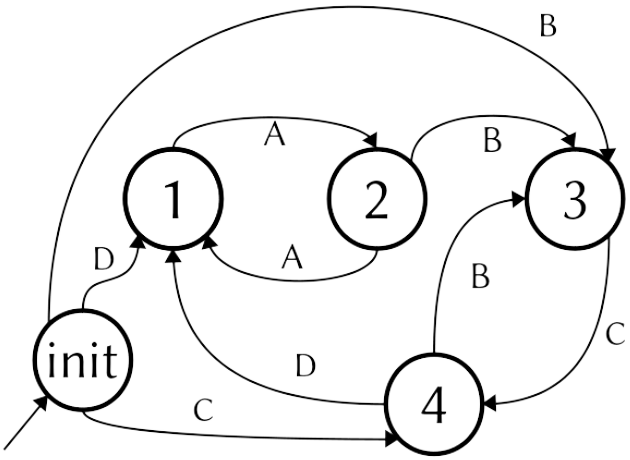


Figure 4.10 The dummy FSM used to evaluate the accuracy of our solution

of the system modelled by the FSM. XML traces are a convenient way of creating a trace by specifying every event (with their event fields) using an XML-based language.

The instances are following the patterns given below:

```
instance 1: BCDAAABCBCDABCDABCBCDABCBCBCDAAABC
instance 2: DAAAAAAAA
instance 3: CDABCBCBCBCDABCBCBCD
```

Overall, our trace is:

```
BCDDAAAABACCDAABCDABCDAABCBCBCBCAAAABCBCDADABCBCBCBCBCBCDDAAABC
```

From the original trace, a script is generating 63 different traces by deleting each of the 63 events in a new trace. The coherence analysis is then applied to every trace and, if inconsistencies were found, we tried to infer missing events.

Inconsistencies were found for 50 traces out of 63. No inconsistency was found when the deleted event was the first event of a given instance, or the last event (it affects 6 traces), or when it was one of the `A` events in the sequence occurring in the instance 2 [4]. This is the expected result, given the definition of an inconsistency. Every inconsistency that was detectable has indeed been found.

4. It corresponds to the cases where the deleted event is 1, 4, 5, 10, 12, 14, 15, 23, 33, 34, 36, 58, or 63

Table 4.4 Evaluation results for one deleted event (Method 1 is using transitions frequencies as weights. Method 2 is unweighted. Method 3 is using the event types frequencies as weights.)

Method	1	2	3
Inferred events	94	86	86
TP (True Positives)	50	50	50
FP (False Positives)	44	36	36
FN (False Negatives)	13	13	13
Accuracy	0.53	0.58	0.58

Out of these 50 traces, the deleted event was successfully recovered in every case. In 31 cases, the inference process was entirely correct, *i.e.*, no additional event was inferred [5]. In 19 cases, there were more inferred events than necessary. Sometimes a path longer than the real path has been computed (*i.e.*, the computed shortest path does not correspond to the actual path if the event had not been deleted, thus creating more inferred events than was necessary). Perhaps this success rate could be improved with better heuristics based on transition weights. In total, 94 events have been inferred: 50 were correct, 44 were extra.

We also wanted to compare our solution with other weighting functions. The first method is the one used in our solution, *i.e.*, using the number of transitions occurrences. In the second method, we simply do not add weights to the arcs of the graph (the weight is 1 for every arc). In the third method, we use the number of occurrences of the event labelling the transition as its weight (instead of using the transitions as in the first method), *i.e.*, the more often an event occurred in a trace, the smaller the weight of a transition labelled with this event.

A summary of the evaluation is given in Table 4.4, where
$accuracy = \frac{TP}{\text{total inferred events}}$ (note that we do not have True Negatives). We can see that the second and third methods are producing identical results. This can be explained by the fact that our state machine is very simple. For every state, there is at most 3 incoming transitions (if we take into account transitions from the initial state), so the number of possible paths is limited. In that case, the results from the second method matched those from the third one, because we tested on a limited number of states and a limited number of events.

We also ran the script to delete 2 consecutive events (Table 4.5), and 3 consecutive events (Table 4.6). We see that the accuracy fluctuates around 0.5 for our solution. As we deleted consecutive events, the sequences of lost events that correspond to a loop in the FSM do

5. It corresponds to the cases where the deleted event is 2, 9, 16, 17, 18, 20, 21, 22, 25, 26, 27, 28, 29, 30, 31, 32, 37, 39, 40, 43, 45, 47, 49, 50, 51, 52, 53, 54, 55, 56, or 62

Table 4.5 Evaluation results for 2 consecutive events deleted

Method	1	2	3
Inferred events	83	67	67
TP (True Positives)	41	41	41
FP (False Positives)	42	26	26
FN (False Negatives)	84	84	84
Accuracy	0.49	0.61	0.61

Table 4.6 Evaluation results for 3 consecutive events deleted

Method	1	2	3
Inferred events	113	95	95
TP (True Positives)	64	64	64
FP (False Positives)	49	31	31
FN (False Negatives)	122	122	122
Accuracy	0.57	0.67	0.67

not lead to an inconsistent state. This is the case for `BC`, `CB` or `DA` sequences. This explains why there are many FN. The example is not very representative of how real systems behave, therefore the heuristics chosen for our solution brings poorer results. Improving the heuristics will be the focus of future work. Nonetheless, the weights are useful to improve the accuracy of the inferred sequence of events, and we can see that the proposed framework allows to find a large number of lost events.

4.6.5 Limits

An important issue that has been considered is the fact that the most likely path may not be the path that has really been taken, especially as losing events may occur mostly in unusual cases. Unfortunately, this is not an easy problem, because it equates to look for "the most likely unlikely case". Thus, we must be aware that the events inferred by our solution may not be perfectly accurate.

As seen in 4.6.3, the overhead is quite big for larger traces (up to 139.56% with the naive algorithm, or up to 30% with the optimized algorithm). This could become problematic when dealing with real use cases. However, this is related to the current implementation of the XML analysis module which was not expecting such an intensive use of its access functions. Work is underway to refactor the XML module and get rid of several inefficiencies. This will

considerably reduce the execution time for our analysis.

4.7 Conclusion and Future Work

In order to handle lost events during trace analysis, we proposed a framework consisting of two parts, the consistency check and the event inference. The former checks every event, as they are sequentially read from the trace, in relation to a state machine for two properties, the inconsistency and the state certainty. The event inference computes a recovery, from every inconsistency found at the preceding step, using Dijkstra's shortest path algorithm. Intervals where the states computed during the analysis are uncertain, as well as any detected inconsistencies, can be highlighted on the views shown to the user.

With this work, we achieved the goal of improving the trace analysis and the understanding of its results, in particular by making the user aware of trace inconsistencies. We were able to answer to the needs of our industrial partner by providing visual clues for diagnosis. Our solution demonstrates that it is feasible to retrieve lost or inaccessible information in a trace, using state machines. Now that we are able to know the certainty of the states and to infer part of the missing information, we could reconstruct an initial global state. This progress is a step towards the parallel analysis of traces, which would greatly improve the capacity of trace analysis.

Further work will focus on increasing the scalability of the analysis. An interesting avenue to investigate is introducing machine learning to compute more accurate probabilities. Moreover, there is a lot of useful information contained in traces and in other analyses results that have not been exploited yet. Finally, as suggested earlier, the automatic computation of the FSM could be implemented, given a set of correct traces for a system.

CHAPITRE 5 DISCUSSION GÉNÉRALE

5.1 Retours sur les résultats obtenus

5.1.1 Résultats de l'évaluation

Dans certains cas, les événements déduits forment une séquence qui n'est pas cohérente avec l'état suivant. Par exemple, lorsque l'on supprime l'événement 13 (un `D` sur l'objet 3), la correction proposée est la séquence `DA`, qui correspond bien à une séquence possible d'événements selon la définition de la FSM correspondante, et qui se trouve être la correction la plus probable à partir des fréquences de transitions. Or, l'événement supprimé se trouvait dans cette partie de la trace : `CDABC...` Donc on voit que la séquence `CDAABC` n'est pas acceptée par la machine à états, c'est une incohérence.

Cela est dû au fait que l'on ne vérifie pas la cohérence *après* avoir effectué la phase d'inférence des événements. Pour prendre en compte ces considérations, il faudrait relancer l'analyse de la cohérence. Actuellement, ce serait très coûteux, car l'analyse ne peut se faire que sur une trace en entier. Si de nombreux allers-retours sont nécessaires, l'analyse va être exécutée sur toute la trace un grand nombre de fois.

De plus, de nombreux événements incohérents détectés dans les traces utilisées pour le test de performance correspondent à des événements de sortie d'appel système (`syscall_exit`) solitaires. Puisqu'on ne peut normalement observer ces événements que lorsque le système est dans l'état suivant, l'observation d'un événement `syscall_entry`, une incohérence est ajoutée à chaque fois. Néanmoins, il arrive souvent que ces événements arrivent avec du retard, ou qu'ils puissent être déclenchés par un mécanisme particulier sans qu'il y ait eu l'entrée correspondante. Nous ne pouvons pas distinguer ces cas d'une incohérence réelle, donc nous devons marquer ces événements comme potentiellement incohérents.

5.2 Précisions sur la solution proposée

5.2.1 Compléments sur l'inférence du contenu des événements

Comme présenté dans l'article, une fois le type d'événement déduit, il est nécessaire d'exploiter le plus d'informations possible afin de déduire son contenu (les champs et leurs valeurs). Pour cela, nous pouvons exploiter les conditions préalables à l'exécution de la transition déduite, conditions qui sont nécessairement vérifiées selon la définition d'une transition. Nous

allons revenir plus en détails sur ce mécanisme.

Dans le cas général, lorsque l'on vérifie si une transition s'applique, chaque condition reliée à cette transition est testée. Il existe trois types de conditions XML :

— DATA : une condition qui porte sur des données du système d'états

— TIME : une condition qui porte sur le temps

— autre : correspond au cas des conditions composées (*or*, *and*, *not*)

Ce sont les conditions de type DATA qui nous sont utiles, c'est ce que l'on veut récupérer pour extraire des informations. Il n'y a rien d'utile dans une condition de type TIME, car l'estampille de temps ne fait pas partie du contenu d'un événement (éventuellement, ce type de conditions pourrait être utilisé pour déduire l'estampille de temps de l'événement perdu). Quant aux autres conditions, elles vont être décomposées par des appels récursifs à la fonction de test, donc les sous-conditions seront éventuellement utilisées si elles sont de type DATA.

Dans certains cas, plusieurs valeurs différents sont possibles pour un même champ d'événement. Par exemple, on peut avoir une condition de la forme suivante :

```
<condition>
  <stateAttribute type="constant" value="Threads" />
  <stateAttribute type="eventField" value="tid" />
  <stateAttribute type="constant" value="Status" />
  <stateValue type="int" value="$PROCESS_STATUS_RUN_USERMODE" />
</condition>
```

Cette condition signifie que l'on cherche à savoir si le fil d'exécution identifié par `tid` est dans l'état "exécution en espace utilisateur" (`$PROCESS_STATUS_RUN_USERMODE`). Dans le contexte de l'inférence du contenu d'événement, on cherche donc à ajouter le champ `tid` au contenu de l'événement déduit ; ce qui signifie que l'on va sélectionner, comme valeurs possibles, toutes les valeurs de `tid` pour lesquelles le fil est dans l'état "exécution en espace utilisateur". Si l'on n'a pas de moyen de discriminer entre ces multiples valeurs, nous nous trouvons dans le cas où le champ est multi-valué.

Une possibilité serait de regarder le passé de chaque fil candidat et de s'en servir pour choisir celui qui est le plus probable (par exemple, un fil où il y a beaucoup d'activité pourrait être le plus probable pour être le fil sélectionné par le système d'exploitation lors de l'ordonnancement sur le CPU).

De plus, un même champ peut se trouver dans plusieurs conditions pour la même transition. Pour chacune de ces conditions, un ensemble de valeurs va être calculé. Lorsqu'un champ a

plusieurs groupes de valeurs possibles, mais que parmi ces groupes, il y en a un contenant seulement une valeur possible, alors on peut dire que cette valeur est la bonne et la sélectionner automatiquement.

Néanmoins, quand plusieurs choix possibles sont de même probabilité, nous avons décidé de laisser l'utilisateur choisir parmi les différentes possibilités qu'on lui propose. En effet, il est le plus à même de faire un choix éclairé, en fonction de ce qu'il connaît du système, ce qu'il voit sur la trace, donc ce qu'il pense être le plus probable. Ainsi, une fenêtre offre à l'utilisateur un moyen interactif de sélectionner parmi les différentes valeurs possibles pour chaque champ multi-valué (voir en Annexes, la Figure B.1). Chaque choix (fait au fur et à mesure) est affiché sur la vue *Global Inference View* et la modifie au fur et à mesure. On enregistre les choix dans l'événement déduit, et éventuellement l'utilisateur peut annuler un choix et revenir en arrière.

5.2.2 Trace contenant les événements déduits

Une fois les événements déduits, il faut pouvoir les intégrer à la trace pour laquelle des événements sont manquants. Malheureusement, les mécanismes internes de Trace Compass ne permettent pas d'insérer des événements à une trace déjà analysée.

Nous avons donc développé un nouveau type de trace, appelé *Inference Trace*. On peut dire que c'est une trace synthétique, car on ne lit et analyse pas une trace réelle. En fait, cette trace enveloppe la trace CTF existante, mais elle possède aussi une référence sur la liste des événements déduits. Ainsi, on peut surcharger la méthode en charge de fournir l'événement suivant d'une trace, afin de celle-ci retourne soit l'événement suivant de la trace réelle, soit l'événement déduit suivant (selon l'itérateur sur ceux-ci) si on a atteint son estampille de temps dans la chronologie. Cette trace nécessite quelques ajustements manuels car ce n'est pas un fonctionnement normalement prévu par Trace Compass, mais elle permet d'effectuer les analyses courantes comme s'il s'agissait d'une trace réelle.

CHAPITRE 6 CONCLUSION

6.1 Synthèse des travaux

Durant ce projet, nous avons conçu une solution permettant de gérer les événements perdus lors de l'analyse de traces et nous l'avons implémentée dans l'outil de visualisation de traces Trace Compass, sous la forme d'un module.

Nous avons été capable de définir une machine à états de telle sorte qu'on puisse construire les structures de données nécessaires à la vérification des événements perdus lors de l'analyse de traces. Des instances de la machine à états sont créées pour chaque objet modélisé et sont identifiées par un attribut unique que l'on peut extraire des événements de la trace.

Pendant l'analyse, nous avons été capable d'identifier correctement des événements incohérents dans une trace, par rapport à la machine à états spécifiée par l'utilisateur. Ceci est rendu possible par l'ajout d'une étape de vérification de la cohérence, lorsqu'un événement ne provoque pas de transition. Il faut contrôler que l'événement ne soit effectivement pas censé provoquer de transition, et non qu'il aurait pu en provoquer une d'un autre état que l'état actuel.

Nous avons également pu démontrer qu'il est possible de retrouver une partie des événements perdus suite à la détection des incohérences. Pour chaque incohérence, nous pouvons utiliser l'algorithme du chemin le plus court de Dijkstra afin de calculer la séquence de transitions entre le dernier état cohérent connu et l'état incohérent correspondant. Ensuite, à partir des informations contenues dans le système d'états, et les conditions qui doivent nécessairement être vérifiées afin que la transition déduite ait lieu, nous pouvons inférer le contenu de l'événement en cours de reconstruction. Cela nous permet de proposer à l'utilisateur des corrections pour la trace étudiée. Enfin, l'ensemble de ces informations est présenté à l'utilisateur sous forme de vues dans Trace Compass.

Les différentes étapes de notre solution sont brièvement récapitulées dans la figure 6.1.

Nous avons donc réussi à améliorer l'analyse de traces en fournissant des informations sur la cohérence des résultats obtenus. Ceci est une indication importante pour tout utilisateur. Notre solution permet aux utilisateurs d'avoir une meilleure compréhension du fonctionnement de leur système. D'un point de vue plus général, ce travail est une preuve de concept qu'il est possible de récupérer des informations perdues, ou inaccessibles, dans une trace, à partir de machines à états finis. C'est un progrès important qui pourra être utilisé pour l'analyse en parallèle des traces, car cela permet de se passer d'un état global initial.

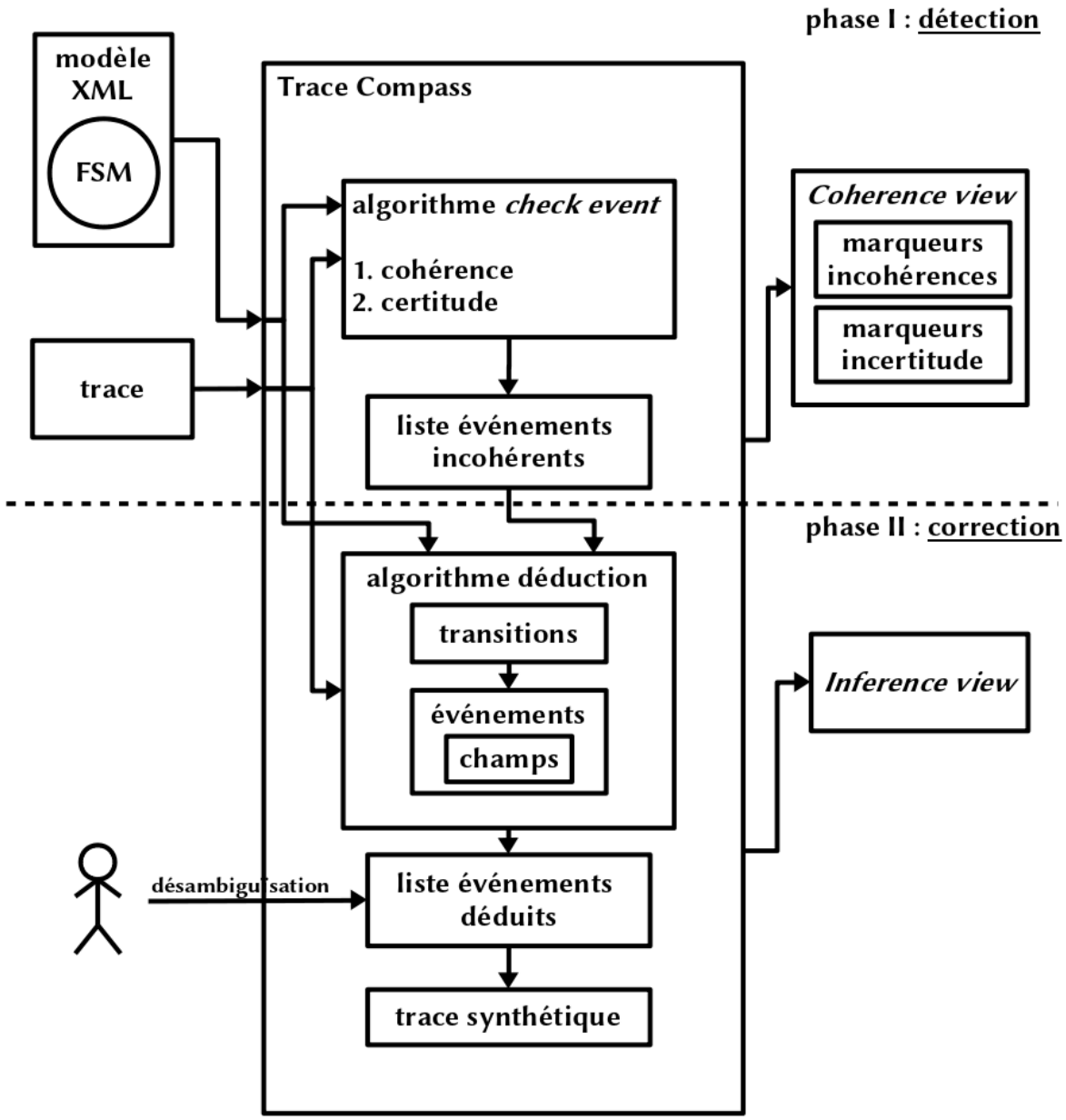


Figure 6.1 Schéma simplifié des étapes suivies lors de l'application de notre solution à une trace

6.2 Limitations de la solution proposée

Une limitation de notre méthode se trouve dans le fait que, s'il n'y a pas d'incohérence après un événement perdu, aucune correction n'est appliquée. Ceci fait sens dans le cas général où l'on cherche à observer la cohérence d'une trace, mais est moins pertinent dans le cas des événements perdus générés par LTTng, où l'on sait qu'il y a bel et bien eu perte d'événements. Même s'il n'y a pas d'incohérence, on pourrait vouloir essayer de reconstruire les événements possiblement manquants, mais cela pose la question de l'identification de ces événements. Une piste envisageable pourrait être de se baser sur des informations temporelles. Par exemple, si l'on ne constate pas de changement d'état pendant toute la période d'événements manquants, et donc que la prochaine transition n'arrive qu'après un délai plus long que d'habitude, on pourrait signaler une incohérence, ce qui activerait par la suite la recherche d'événements manquants. Ceci pourrait consistuer une option de vérification "agressive" (c'est-à-dire en allégeant les hypothèses) des incohérences. Si on n'a pas détecté que des événements avaient été perdus sans mener à une incohérence, ce n'est pas très grave car l'analyse n'est pas vraiment faussée, puisque l'on n'a pas calculé quelque chose de faux. Il nous manque juste quelques changements d'états ou statistiques, mais sans que cela ne nuise vraiment à la compréhension globale.

Il est important de souligner que, dans le cas des événements perdus, le chemin le plus probable ne correspond peut-être pas à ce qui s'est effectivement passé. S'il y a eu des événements perdus, c'est peut-être parce qu'il s'est passé quelque chose d'improbable (par exemple, un très grand nombre d'opérations d'ordonnancement ou d'appels système d'un coup, ce qui est normalement peu probable). Ce fait pourrait être une limitation de la solution. Néanmois, nous ne pouvons que souligner ce risque, car il n'y a pas de solution pour traiter ce problème. En effet, il semble difficile de trouver l'improbabilité la plus probable.

Un problème pratique pouvant limiter l'applicabilité de notre solution est la mauvaise mise à l'échelle dont souffre la solution pour le moment. En effet, pour de grandes traces, le surcoût de la méthode devient important si l'on active la vérification pour toute la trace. Ceci pose la question de savoir s'il est pertinent de vérifier la cohérence d'une trace au complet. Toujours est-il que le travail de réécriture du module des analyses XML qui est en cours devrait s'intéresser à cette problématique, ce qui devrait permettre des gains de performance pour notre solution qui fait un usage important des mécanismes de ce module.

6.3 Améliorations futures

Notre solution nécessite d'avoir accès à la définition de la machine à états modélisant le système considéré. Dans l'implémentation actuelle de cette solution, nous utilisons les machines à états XML, qui nécessitent d'être définies par l'utilisateur. On pourrait envisager d'automatiser la définitions des FSMs, comme mentionné dans 2.3.1, à partir d'un ensemble de traces pour notre système. Plus pertinent encore, il faut rappeler que Trace Compass définit déjà des machines à états dans ses modules d'analyse, afin de transformer les événements en changements d'états. Il serait donc possible d'utiliser ces définitions, sans passer par le module XML, qu'on utilise pour le moment à des fins de prototypage.

Dans le contexte de l'application de notre solution à Trace Compass, il serait nécessaire d'améliorer les performances, afin de pouvoir travailler avec des traces de volume industriel. Notamment, il serait nécessaire d'adapter certains des mécanismes de Trace Compass, qui n'avaient pas nécessairement été conçus pour notre cas d'utilisation. Il serait particulièrement pertinent de faire en sorte que les analyses ne se fassent pas en un seul coup sur toute la trace, puisqu'on voudrait parfois refaire l'analyse sur une petite portion seulement (par exemple, un certain intervalle de temps après celui où l'on a proposé des événements déduits).

Afin d'améliorer l'exactitude des événements déduits, des travaux futurs pourraient se concentrer sur l'introduction de l'apprentissage machine pour calculer des probabilités plus précises. En effet, il y a de nombreuses statistiques contenues dans la trace qui sont encore inexploitées. Avec un modèle bien défini comme les machines à états, il est possible d'y appliquer des méthodes d'apprentissage afin d'enrichir la qualité des corrections. De plus, il serait intéressant de prendre en compte d'autres éléments, comme le temps entre les transitions, la recherche de motifs d'événements (pairs, triplés d'événements que l'on retrouve souvent, etc.).

RÉFÉRENCES

"Event tracing", Microsoft Developer Network, accessed : 2018-06-15. En ligne : https://msdn.microsoft.com/library/bb968803.aspx

A. Adriansyah, B. F. van Dongen, et W. M. van der Aalst, "Conformance checking using cost-based fitness analysis", dans *Enterprise Distributed Object Computing Conference (EDOC), 2011 15th IEEE International*. IEEE, 2011, pp. 55–64.

A. V. Aho et T. G. Peterson, "A Minimum Distance Error-Correcting Parser for Context-Free Languages", *SIAM Journal on Computing*, 1972. DOI : 10.1137/0201022. En ligne : http://epubs.siam.org/doi/abs/10.1137/0201022

A. V. Aho, R. Sethi, et J. D. Ullman, *Compilers : Principles, Techniques, and Tools*. Addison-Wesley, 1986.

P. D. Allison, *Missing data*. Sage Thousand Oaks, CA, 2012.

J. C. Amengual et E. Vidal, "Efficient error-correcting Viterbi parsing", *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 20, no. 10, pp. 1109–1116, Oct. 1998. DOI : 10.1109/34.722628

S. O. Anderson, R. C. Backhouse, E. H. Bugge, et C. P. Stirling, "An Assessment of Locally Least-Cost Error Recovery", *The Computer Journal*, vol. 26, no. 1, pp. 15–24, Jan. 1983. DOI : 10.1093/comjnl/26.1.15. En ligne : https://academic.oup.com/comjnl/article/26/1/15/457036

P. Boullier et M. Jourdan, "A new error repair and recovery scheme for lexical and syntactic analysis", *Science of Computer Programming*, vol. 9, no. 3, pp. 271–286, Déc. 1987. DOI : 10.1016/0167-6423(87)90010-4. En ligne : http://www.sciencedirect.com/science/article/pii/0167642387900104

A. Bouloutas, G. W. Hart, et M. Schwartz, "Two extensions of the Viterbi algorithm", *IEEE Transactions on Information Theory*, vol. 37, no. 2, pp. 430–436, Mars 1991. DOI : 10.1109/18.75270

R. Braden, "Requirements for Internet Hosts – Communication Layers", Internet Engineering Task Force, RFC 1122, October 1989. En ligne : https://tools.ietf.org/html/rfc1122

T. Casavanf, “Recovering uncorrupted event traces from corrupted event traces in parallel/distributed computing systems”, dans *Proceedings 20th International Conference Parallel Processing 1991*, vol. 2. CRC Press, 1991, p. 108. En ligne : https://books.google.ca/books?hl=fr&lr=&id=BIctFWjqNuwC&oi=fnd&pg=SL8-PA108&dq=Recovering+uncorrupted+event+traces+from+corrupted+event+traces+in+parallel/distributed+systems&ots=QR53ELh8LK&sig=PNxGmVAgACrOk_eazLHxDJhLgiM

C. G. Cassandras et S. Lafortune, *Introduction to discrete event systems*, 2e éd. New York : Springer, 2008.

T. H. Cormen, C. E. Leiserson, R. L. Rivest, et C. Stein, *Introduction to algorithms*. MIT press, 2009.

J. A. Dain, “A practical minimum distance method for syntax error handling”, *Computer Languages*, vol. 20, no. 4, pp. 239–252, Nov. 1994. DOI : 10.1016/0096-0551(94)90006-X. En ligne : http://www.sciencedirect.com/science/article/pii/009605519490006X

M. De Leoni, F. M. Maggi, et W. M. van der Aalst, “Aligning Event Logs and Declarative Process Models for Conformance Checking.” dans *BPM*, vol. 12. Springer, 2012, pp. 82–97. En ligne : http://link.springer.com/content/pdf/10.1007/978-3-642-32885-5.pdf#page=95

A. C. De Melo, “The new linux’perf’tools”, dans *Slides from Linux Kongress*, vol. 18, 2010.

M. Desnoyers, “Common trace format (ctf) specification (v1.8.2)”, 2010. En ligne : http://diamon.org/ctf/#specification

——, “Low-impact operating system tracing”, Thèse de doctorat, École Polytechnique de Montréal, 2009.

M. Desnoyers et M. R. Dagenais, “The lttng tracer : A low impact performance and behavior monitor for gnu/linux”, dans *OLS (Ottawa Linux Symposium)*, série Proceedings of the Linux Symposium, vol. 1, july 2006, pp. 209–224.

E. W. Dijkstra, “A note on two problems in connexion with graphs”, *Numerische Mathematik*, vol. 1, no. 1, pp. 269–271, Déc. 1959. DOI : 10.1007/BF01386390. En ligne : https://link.springer.com/article/10.1007/BF01386390

J. Earley, “An Efficient Context-free Parsing Algorithm”, *Commun. ACM*, vol. 13, no. 2, pp. 94–102, Fév. 1970. DOI : 10.1145/362007.362035. En ligne : http://doi.acm.org/10.1145/362007.362035

N. Ezzati-Jivan et M. R. Dagenais, “A Stateful Approach to Generate Synthetic Events from Kernel Traces”, *Adv. Soft. Eng.*, vol. 2012, pp. 6 :6–6 :6, Jan. 2012. DOI : 10.1155/2012/140368. En ligne : http://dx.doi.org/10.1155/2012/140368

E. Fabre, “Diagnosis and Automata”, dans *Control of Discrete-Event Systems*, série Lecture Notes in Control and Information Sciences. Springer, London, 2013, pp. 85–106. En ligne : https://link.springer.com/chapter/10.1007/978-1-4471-4276-8_5

E. Fabre et L. Jezequel, “On the construction of probabilistic diagnosers”, *IFAC Proceedings Volumes*, vol. 43, no. 12, pp. 229–234, Jan. 2010. DOI : 10.3182/20100830-3-DE-4013.00039. En ligne : http://www.sciencedirect.com/science/article/pii/S1474667015324617

R. Fahem, “Points de trace statiques et dynamiques en mode noyau”, M.Sc.A. En ligne : https://search.proquest.com/docview/1080972601/abstract/945AFB431D1A4262PQ/1

P.-M. Fournier et M. R. Dagenais, “Analyzing blocking to debug performance problems on multi-core systems”, vol. 44, no. 2, pp. 77–87. DOI : 10.1145/1773912.1773932. En ligne : http://doi.acm.org/10.1145/1773912.1773932

P. J. García-Laencina, J.-L. Sancho-Gómez, et A. R. Figueiras-Vidal, “Pattern classification with missing data : a review”, *Neural Computing and Applications*, vol. 19, no. 2, pp. 263–282, 2010.

M. Gebai et M. R. Dagenais, “Survey and analysis of kernel and userspace tracers on linux : Design, implementation, and overhead”, vol. 51, no. 2, pp. 26 :1–26 :33, 3 2018. DOI : 10.1145/3158644. En ligne : http://doi.acm.org/10.1145/3158644

F. Giraldeau, J. Desfossez, D. Goulet, M. Dagenais, et M. Desnoyers, “Recovering system metrics from kernel trace”, dans *Linux Symposium*, vol. 109, 2011.

P. Gladyshev et A. Patel, “Finite state machine approach to digital event reconstruction”, *Digital Investigation*, vol. 1, no. 2, pp. 130–149, Juin 2004. DOI : 10.1016/j.diin.2004.03.001. En ligne : http://www.sciencedirect.com/science/article/pii/S1742287604000271

D. Goulet, “Unified kernel/user-space efficient linux tracing architecture”, Mémoire de maîtrise, École Polytechnique de Montréal, Avril 2012.

S. L. Graham et S. P. Rhodes, “Practical Syntactic Error Recovery”, *Commun. ACM*, vol. 18, no. 11, pp. 639–650, Nov. 1975. DOI : 10.1145/361219.361223. En ligne : http://doi.acm.org/10.1145/361219.361223

B. Gregg, “Linux performance analysis and tools”, Technical report, Joyent, 2013.

F. Halsall, *Computer Networking and the Internet*, 5e éd. Addison Wesley, Jan 2005.

R. W. Hamming, “Error detecting and error correcting codes”, vol. 29, no. 2, pp. 147–160. DOI : 10.1002/j.1538-7305.1950.tb00463.x

K. Koskimies et E. Mäkinen, “Automatic synthesis of state machines from trace diagrams”, *Software : Practice and Experience*, vol. 24, no. 7, pp. 643–658, Juil. 1994. DOI : 10.1002/spe.4380240704. En ligne : http://onlinelibrary.wiley.com/doi/10.1002/spe.4380240704/abstract

K. Kouame, N. Ezzati-Jivan, et M. R. Dagenais, “A flexible data-driven approach for execution trace filtering”, dans *2015 IEEE International Congress on Big Data*, pp. 698–703. DOI : 10.1109/BigDataCongress.2015.112

K. G. Kouamé, “Langage dédié et analyse automatisée pour la détection de patrons au sein de traces d’exécution”, Mémoire de maîtrise, École Polytechnique de Montréal, August 2015.

S.-Y. Lu et K.-S. Fu, “Stochastic Error-Correcting Syntax Analysis for Recognition of Noisy Patterns”, *IEEE Transactions on Computers*, vol. C-26, no. 12, pp. 1268–1276, Déc. 1977. DOI : 10.1109/TC.1977.1674788

J. Lunze et J. Schröder, “State Observation and Diagnosis of Discrete-Event Systems Described by Stochastic Automata”, *Discrete Event Dynamic Systems*, vol. 11, no. 4, pp. 319–369, Oct. 2001. DOI : 10.1023/A:1011273108731. En ligne : https://link.springer.com/article/10.1023/A:1011273108731

R. D. Masellis, C. D. Francescomarino, C. Ghidini, et S. Tessaris, “Enhancing Workflow-Nets with Data for Trace Completion”, dans *Business Process Management Workshops*, série Lecture Notes in Business Information Processing. Springer, Cham, Sep. 2017, pp. 89–106. En ligne : https://link.springer.com/chapter/10.1007/978-3-319-74030-0_6

G. Matni et M. Dagenais, “Automata-based approach for kernel trace analysis”, dans *Electrical and Computer Engineering, 2009. CCECE ’09. Canadian Conference on.* IEEE, 2009.

H. Nemati, S. D. Sharma, et M. R. Dagenais, “Fine-grained nested virtual machine performance analysis through first level hypervisor tracing”, dans *Proceedings of the 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, série CCGrid ’17. IEEE Press, pp. 84–89. DOI : 10.1109/CCGRID.2017.20. En ligne : https://doi.org/10.1109/CCGRID.2017.20

I. S. I. U. of Southern California, “TRANSMISSION CONTROL PROTOCOL – DARPA INTERNET PROGRAM PROTOCOL SPECIFICATION”, RFC 793, September 1981. En ligne : https://tools.ietf.org/html/rfc793

C. Perkins, O. Hodson, et V. Hardman, “A survey of packet loss recovery techniques for streaming audio”, vol. 12, no. 5, pp. 40–48. DOI : 10.1109/65.730750

W. W. Peterson et D. T. Brown, “Cyclic codes for error detection”, vol. 49, no. 1, pp. 228–235, 01 1961. DOI : 10.1109/JRPROC.1961.287814

F. Reumont-Locke, “Méthodes efficaces de parallélisation de l’analyse de traces noyau”, Mémoire de maîtrise, École Polytechnique de Montréal, August 2015.

A. Rogge-Solti, R. S. Mans, W. M. van der Aalst, et M. Weske, “Improving documentation by repairing event logs”, dans *IFIP Working Conference on The Practice of Enterprise Modeling*. Springer, 2013, pp. 129–144.

S. Rostedt, “ftrace - function tracer”, 2008. En ligne : https://www.kernel.org/doc/Documentation/trace/ftrace.txt

A. Rozinat et W. M. P. van der Aalst, “Conformance checking of processes based on monitoring real behavior”, *Information Systems*, vol. 33, no. 1, pp. 64–95, Mars 2008. DOI : 10.1016/j.is.2007.07.001. En ligne : http://www.sciencedirect.com/science/article/pii/S030643790700049X

M. S. Ryan et G. R. Nudd, “The Viterbi algorithm”, University of Warwick. Department of Computer Science, Report Number 238, Fév. 1993. En ligne : http://wrap.warwick.ac.uk/60926/

M. Sampath, R. Sengupta, S. Lafortune, K. Sinnamohideen, et D. C. Teneketzis, “Failure diagnosis using discrete-event models”, *IEEE Transactions on Control Systems Technology*, vol. 4, no. 2, pp. 105–124, Mars 1996. DOI : 10.1109/87.486338

M. Sayed-Mouchaweh, “Decentralized Fault Free Model Approach for Fault Detection and Isolation of Discrete Event Systems”, *European Journal of Control*, vol. 18,

no. 1, pp. 82–93, Jan. 2012. DOI : 10.3166/ejc.18.82-93. En ligne : http://www.sciencedirect.com/science/article/pii/S0947358012705290

J. L. Schafer et J. W. Graham, “Missing data : our view of the state of the art.” *Psychological methods*, vol. 7, no. 2, p. 147, 2002.

W. Song, X. Xia, H. A. Jacobsen, P. Zhang, et H. Hu, “Heuristic Recovery of Missing Events in Process Logs”, dans *2015 IEEE International Conference on Web Services*, Juin 2015, pp. 105–112. DOI : 10.1109/ICWS.2015.24

——, “Efficient Alignment Between Event Logs and Process Models”, *IEEE Transactions on Services Computing*, vol. 10, no. 1, pp. 136–149, Jan. 2017. DOI : 10.1109/TSC.2016.2601094

R. A. Thompson, “Language correction using probabilistic grammars”, vol. C-25, no. 3, pp. 275–286, 03 1976. DOI : 10.1109/TC.1976.5009254

D. Toupin, “Using tracing to diagnose or monitor systems”, *IEEE software*, vol. 28, no. 1, pp. 87–91, 2011.

H. M. W. Verbeek, T. Basten, V. D. Aalst, et W. M. P, “Diagnosing Workflow Processes using Woflan”, *The Computer Journal*, vol. 44, no. 4, pp. 246–279, Jan. 2001. DOI : 10.1093/comjnl/44.4.246. En ligne : https://academic.oup.com/comjnl/article/44/4/246/357862/Diagnosing-Workflow-Processes-using-Woflan

A. Viterbi, “Error bounds for convolutional codes and an asymptotically optimum decoding algorithm”, vol. 13, no. 2, pp. 260–269, 04 1967. DOI : 10.1109/TIT.1967.1054010

J. Wang, S. Song, X. Zhu, et X. Lin, “Efficient Recovery of Missing Events”, *Proc. VLDB Endow.*, vol. 6, no. 10, pp. 841–852, Août 2013. DOI : 10.14778/2536206.2536212. En ligne : http://dx.doi.org/10.14778/2536206.2536212

F. Wininger, “Conception flexible d’analyses issues d’une trace système”, Mémoire de maîtrise, École Polytechnique de Montréal, April 2014.

F. Wininger, N. Ezzati-Jivan, et M. R. Dagenais, “A declarative framework for stateful analysis of execution traces”, *Software Quality Journal*, vol. 25, no. 1, pp. 201–229, Mar 2017. DOI : 10.1007/s11219-016-9311-0. En ligne : https://doi.org/10.1007/s11219-016-9311-0

I.-S. Yun, K.-M. Choe, et T. Han, “Syntactic error repair using repair patterns”, *Information Processing Letters*, vol. 47, no. 4, pp. 189–196, Sep. 1993. DOI : 10.1016/0020-0190(93)90031-4. En ligne : http://www.sciencedirect.com/science/article/pii/0020019093900314

J. Zaytoon et S. Lafortune, “Overview of fault diagnosis methods for discrete event systems”, vol. 37, no. 2, pp. 308–320, 12 2013. DOI : 10.1016/j.arcontrol.2013.09.009. En ligne : http://www.sciencedirect.com/science/article/pii/S1367578813000552

J. Zaytoon et M. Sayed-Mouchaweh, “Discussion on fault diagnosis methods of discrete event systems”, *IFAC Proceedings Volumes*, vol. 45, no. 29, pp. 9–12, 2012.

ANNEXE A PREUVE DE LA NON-DIAGNOSTICABILITÉ D'UN SYSTÈME TRACÉ

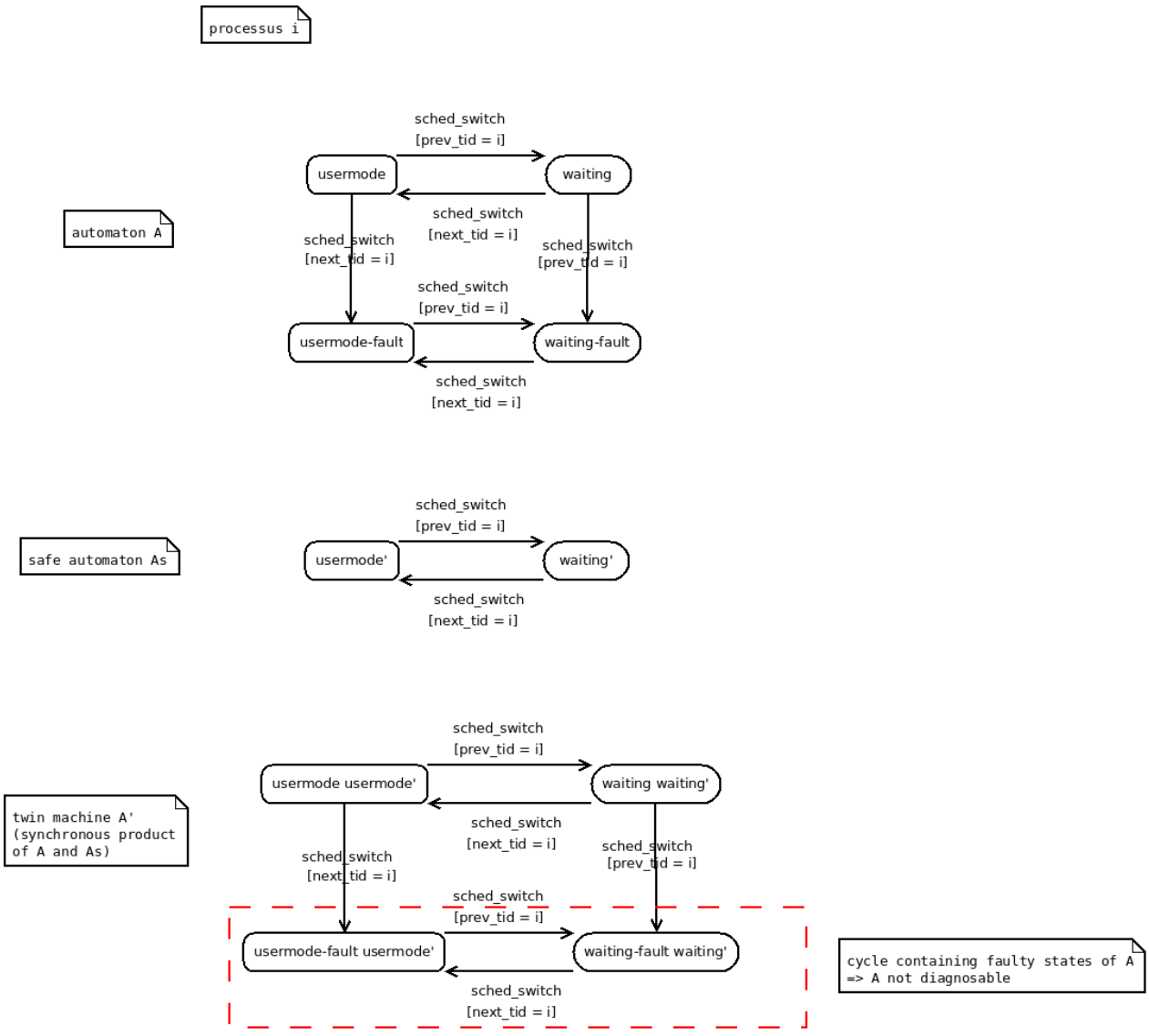


Figure A.1 Preuve de la non diagnosticabilité d'un système typique étudié par la méthode de la machine jumelée

ANNEXE B ILLUSTRATION DE LA SÉLECTION D'UNE VALEUR POUR UN CHAMP MULTI-VALUÉ

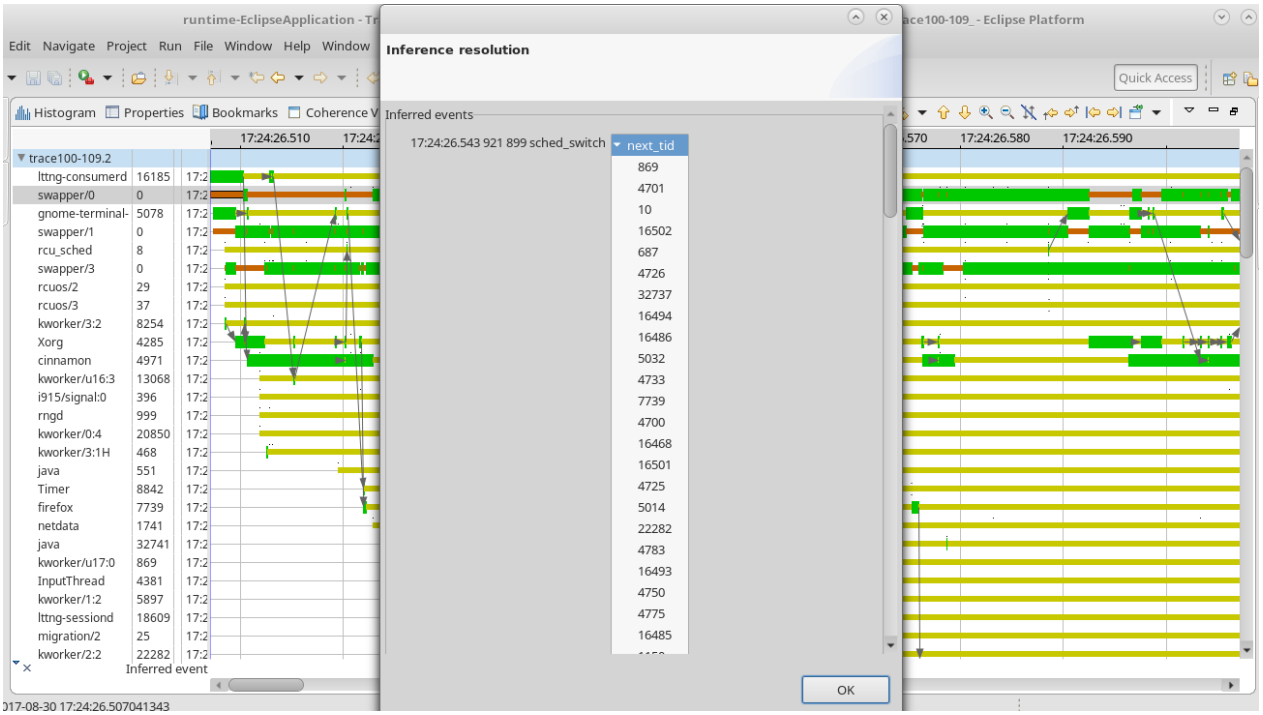


Figure B.1 Fenêtre permettant de sélectionner une valeur pour un champ parmi un ensemble de valeurs déduites de même probabilité